Building IDL Applications

RSI
Research Systems Inc.

IDL Version 6.0
July, 2003 Edition

0703IDL60BLD

Restricted Rights Notice

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

Limitation of Warranty

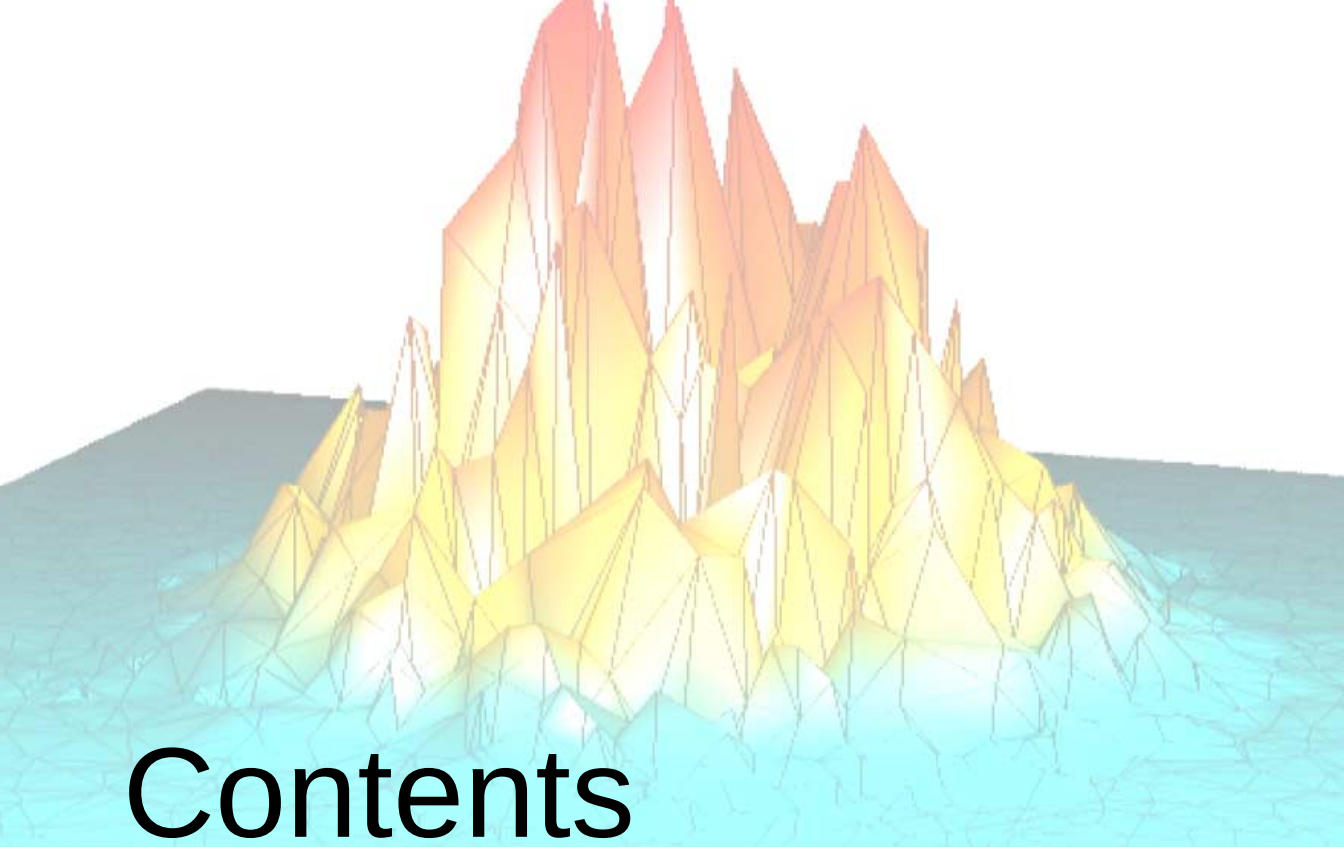
Research Systems, Inc. makes no warranties, either express or implied, as to any matter not expressly set forth in the license agreement, including without limitation the condition of the software, merchantability, or fitness for any particular purpose.

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

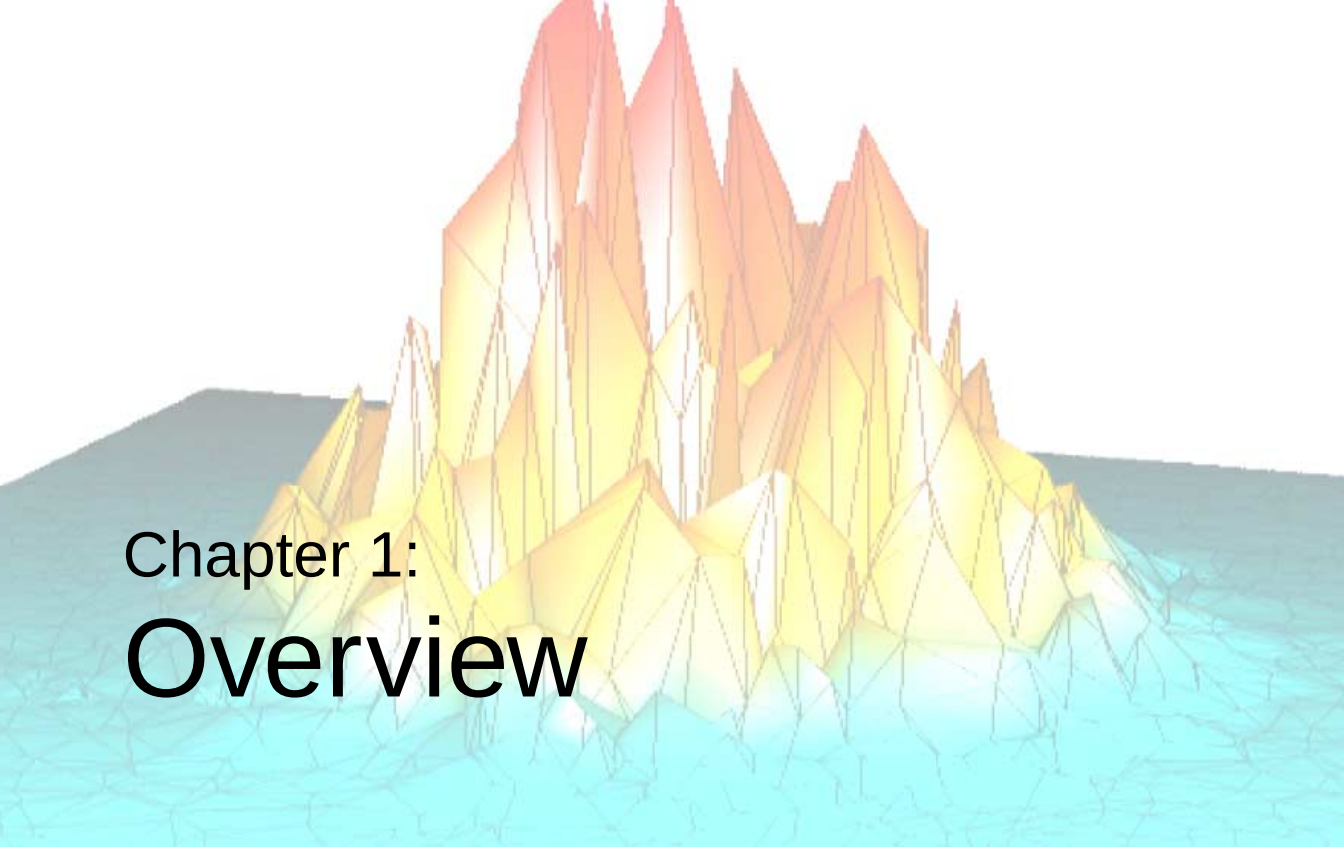
Permission to Reproduce this Manual

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

Acknowledgments



Contents



Chapter 1:
Overview

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” in the *IDL Reference Guide* manual and “Tips on Creating Widget Applications” on page 809 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.

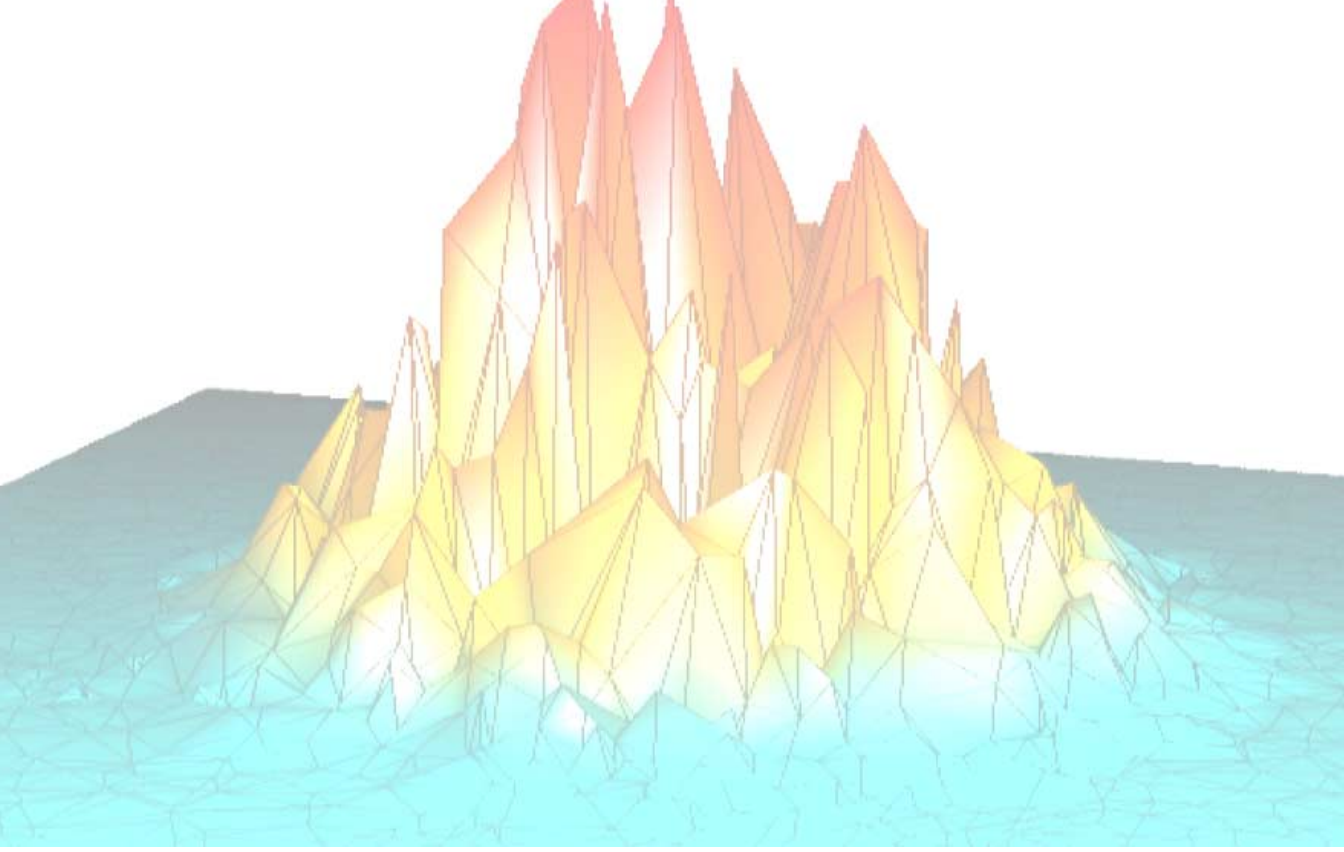
If you would like to distribute your IDL application to people who do not have access to an IDL license, you have several options. Many IDL applications will run in the freely-available IDL Virtual Machine. If your application uses features not available in the virtual machine, you may wish to consider a *runtime IDL* licensing agreement. Runtime IDL licenses allow you to distribute a special version of IDL along with your application. See Chapter 21, “Distributing IDL Applications” for a complete discussion of the different ways you can distribute an application written in IDL.

About Building Applications in IDL

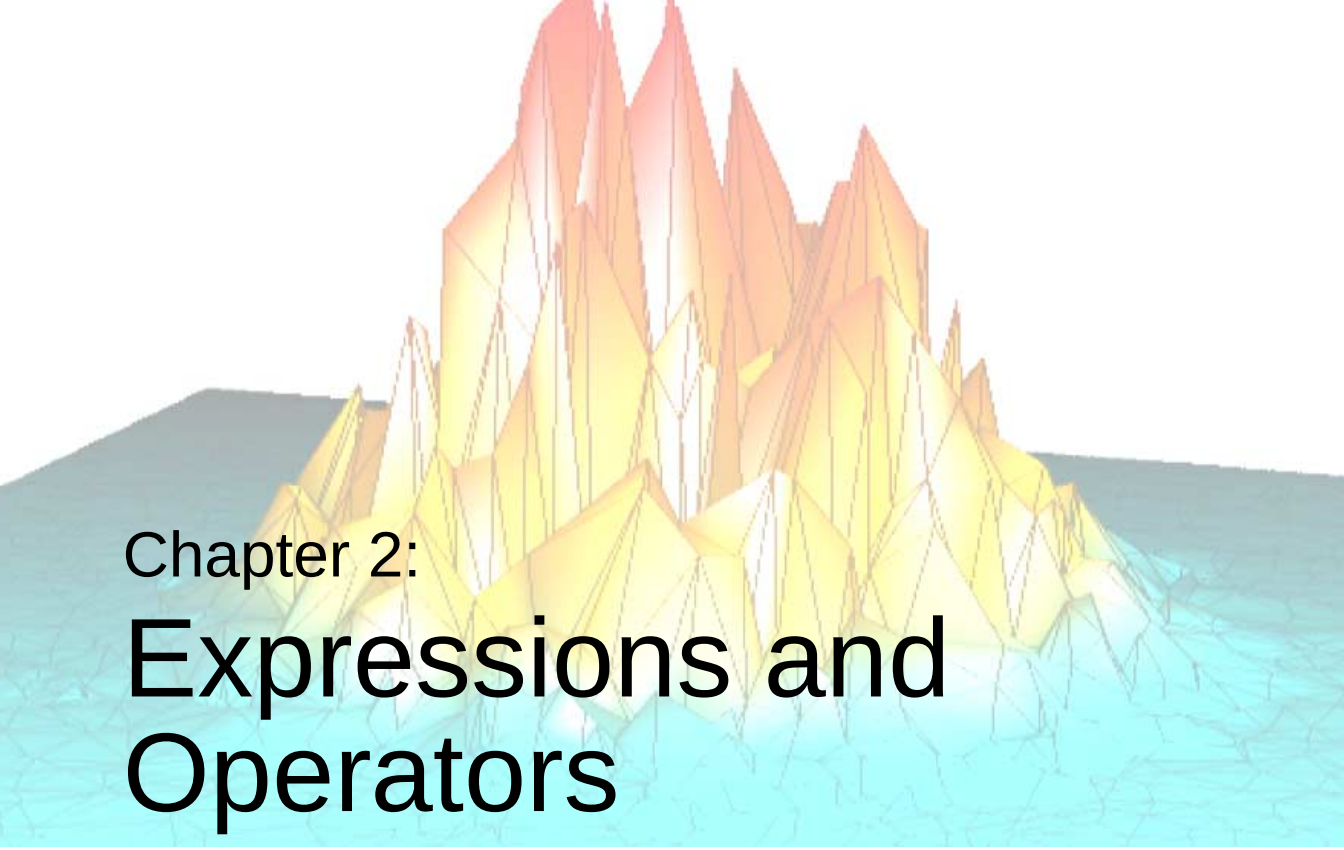
IDL is a complete computing environment for the interactive analysis and visualization of data. IDL integrates a powerful, array-oriented language with numerous mathematical analysis and graphical display techniques. Programming in IDL is a time-saving alternative to programming in FORTRAN or C—using 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”.



Part I: Components of the IDL Language



Chapter 2:

Expressions and Operators

The following topics are covered in this chapter:

Overview

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 π, then applies the trigonometric sine function. This result can be used as an operand to form a more complex expression or as an argument to yet another function (e.g., EXP(SIN(A*!PI)) evaluates $e^{sin\ \pi a}$).

IDL Operators

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

- Parentheses
- Square Brackets
- Mathematical Operators
- Minimum and Maximum Operators
- Matrix Multiplication
- Array Concatenation
- Logical Operators
- Bitwise Operators
- Relational Operators

Parentheses

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

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

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

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

Square Brackets

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

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

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

Note

In versions of IDL prior to version 5.0, parentheses were used to enclose array subscripts. While using parentheses to enclose array subscripts will continue to work as in previous version of IDL, we strongly suggest that you use brackets in all new code. See “Array Subscript Syntax: [ ] vs. ( )” on page 128 for additional details.

Mathematical Operators

There are seven basic IDL mathematical operators, described below.

Assignment

The equal sign (=) is the assignment operator. The value of the expression on the right hand side of the equal sign is stored in the variable, subscript element, or range on the left side. For example, the following assigns the value 32 to A.

```
A = 32
```

See Chapter 11, “Assignment” for more information.

Compound Assignment Operators

In addition to the standard assignment operator, there are numerous *compound operators* (+=, -=, *etc*.) that combine assignment with another operator. Compound assignment operators provide succinct syntax for expressions in which the same variable would otherwise be present on both sides of the equal sign. For example, the following statements both add 100 to the current value of the variable A:

```
A = A + 100
A += 100
```

See “Compound Assignment Operators” in Chapter 11 for more information and a list of compound assignment operators.

Addition

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

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

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

Subtraction and Negation

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

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

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

Multiplication

The asterisk (*) is the multiplication operator. For example:

```
; Store the product of 2 and 5 in variable C:
C = 2 * 5
```

Division

The forward slash (/) is the division operator. For example:

```
; Store the result of 10.0 divided by 3.2 in variable D:
D = 10.0/3.2
```

Exponentiation

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

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 $\ln(A) = \ln(re^{i\theta}) = \ln(r) + i\theta$.

Modulo

The keyword MOD is the modulo operator. 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. For example:

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

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

Increment/Decrement

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 evaluated 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.

Minimum and Maximum Operators

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

The Minimum Operator

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

```
;Set A equal to 3.
A = 5 < 3

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

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

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

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

For complex numbers the absolute value (or modulus) is used to determine which value is smaller. If both values have the same magnitude then the first value is returned.

For example:

```
; 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)
```

The Maximum Operator

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

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

;Plot positive points only. Negative points are plotted as zero.
```

```
PLOT, ARR > 0
```

For complex numbers the absolute value (or modulus) is used to determine which value is larger. If both values have the same magnitude then the first value is returned. For example:

```
; 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)
```

Matrix Multiplication

IDL has two operators used to multiply arrays and matrices.

The # Operator

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

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.

The ## Operator

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

For an example illustrating the difference between the two, see “Multiplying Arrays” in Chapter 22 of the *Using IDL* manual.

Array Concatenation

The square brackets are used as array concatenation operators. Operands enclosed in square brackets and separated by commas are concatenated to form larger arrays. The

expression [A,B] is an array formed by concatenating A and B, which can be scalars or arrays, along the first dimension.

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

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

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

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

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

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

Note

Array concatenation is a relatively inefficient operation, and should only be performed once for a given set of data if possible.

Logical Operators

There are three logical operators in IDL: &&, ||, and ~.

&&

The logical && operator performs the logical short-circuiting “and” operation on two scalars or one-element arrays, returning 1 if both operands are true and 0 if either operand is false.

||

The logical || operator performs the logical short-circuiting “or” operation on two scalars or one-element arrays, returning 1 if either of the operands is true and 0 if both are false.

~

The logical ~ operator performs the logical “not” operation on a scalar or array operand. If the operand is a scalar, it returns scalar 1 if the operand is false or scalar 0 if the operand is true. If the operand is an array, it returns an array containing a 1 for each element of the operand array that is false, and a 0 for each element that is true.

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.

When is an Operand True?

When evaluated by a logical operator, an expression is considered to be “true” under the following conditions:

- For numerical operands, if the value is non-zero.
- For string operands, if the value is non-null.
- For heap variables (pointers and object references), if the value is non-null.

Short-circuiting

The `&&` and `||` logical operators 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.

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)
```

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.

AND

The bitwise AND operator performs the logical “and” operation on two scalar or array operands. 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. The returned values are as follows:

- For integer operands, AND performs a bitwise “and” operation and returns the result. For example, the statement

  ```
  5 AND 6 = 4
  ```

  is represented in binary as follows:

  ```
  0101 AND 0110 = 0100
  ```

- For floating-point and complex operands, AND returns the second operand if the first operand is not equal to zero; otherwise, the returned value is zero.
- For string operands, AND returns the second operand if the first operand is non-null; otherwise, the returned value is the null string.
- The bitwise AND operator is not valid for heap variable operands.

NOT

The bitwise NOT operator returns the bitwise inverse of its scalar or array operand. If the operand is a scalar, it returns a scalar value. If the operand is an array, it returns an

array containing one value for each element of the operand array. The returned values are as follows:

- For integer operands, NOT returns the complement of each bit of the operand. For example, the statement

  ```
  NOT 4 = -5
  ```

  is represented in binary as follows:

  ```
  NOT 0100 = 1011
  ```

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

Warning

Use caution when using the return value from the bitwise NOT operator as an operand for the logical operators && and ||. See “Note on the NOT Operator” on page 33 for additional discussion.

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.

OR

The bitwise OR operator performs the logical “inclusive or” operation on two scalar or array operands. 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. The returned values are as follows:

- For integer operands, OR performs a bitwise inclusive “or” operation and returns the result. For example, the statement

  ```
  3 OR 5 equals 7
  ```

  is represented in binary as follows:

  ```
  0011 OR 0101 = 0111
  ```

- For floating-point and complex operands, OR returns the first operand if it is non-zero, or the second operand otherwise.

- For string operands, OR returns the first operand if it is non-null, or the second operand otherwise.
- The bitwise NOT operator is not valid for heap variable operands.

XOR

The bitwise XOR operator performs the logical “exclusive or” operation on two scalar or array operands. 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 operand array. The returned values are as follows:

- 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. For example, the statement:

  ```
  4 XOR 3 = 7
  ```

  is represented in binary as follows:

  ```
  0100 XOR 0011 = 0111
  ```

- The bitwise XOR operator is not valid for other types.

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
```

Note on 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
```

Relational Operators

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

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

Table 2-1: Relational Operators

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

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

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

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

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

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

EQ

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

For complex numbers both the real and imaginary parts must be equal. For example:

print, COMPLEX(1,2) EQ COMPLEX(1,2) ; returns true

print, COMPLEX(1,2) EQ COMPLEX(1,-2) ; returns false

NE

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

For complex numbers both the real and imaginary parts must be equal to return a false value. For example:

print, COMPLEX(1,2) NE COMPLEX(1,2) ; returns false

print, COMPLEX(1,2) NE COMPLEX(1,-2) ; returns true

GE

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

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

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

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

For complex numbers the absolute value (or modulus) is used for the comparison.

GT

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

For complex numbers the absolute value (or modulus) is used for the comparison.

LE

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

For complex numbers the absolute value (or modulus) is used for the comparison.

LT

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

For complex numbers the absolute value (or modulus) is used for the comparison.

Using Relational Operators with Arrays

Relational operators can be applied to arrays, and the resulting array of ones and zeroes can be used as an operand. For example, the expression, ARR * (ARR LE 100) is an array equal to ARR except that all points greater than 100 have been reduced to zero. The expression (ARR LE 100) is an array that contains a 1 where the corresponding element of ARR is less than or equal to 100, and zero otherwise. For example, to print the number of positive elements in the array ARR:

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

Using Relational Operators with Infinity and NaN Values

On Windows and Solaris x86 platforms, 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” in the *IDL Reference Guide* manual and “Special Floating-Point Values” on page 434.

Conditional Expression

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 335 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

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

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)
Fifth	+ (addition)
	- (subtraction and negation)
	< (minimum)
	> (maximum)
	NOT (Bitwise negation)

Table 2-2: Operator Precedence

Priority	Operator
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)
	~ (Logical negation)
Ninth	?: (conditional expression)

Table 2-2: (Continued) Operator Precedence

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 2-3: Examples of Expressions

Data Type and Structure of Expressions

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

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

The *structure of an expression* 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. 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.

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.

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 + 1 = 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 2-4: Structure of Expressions

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

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

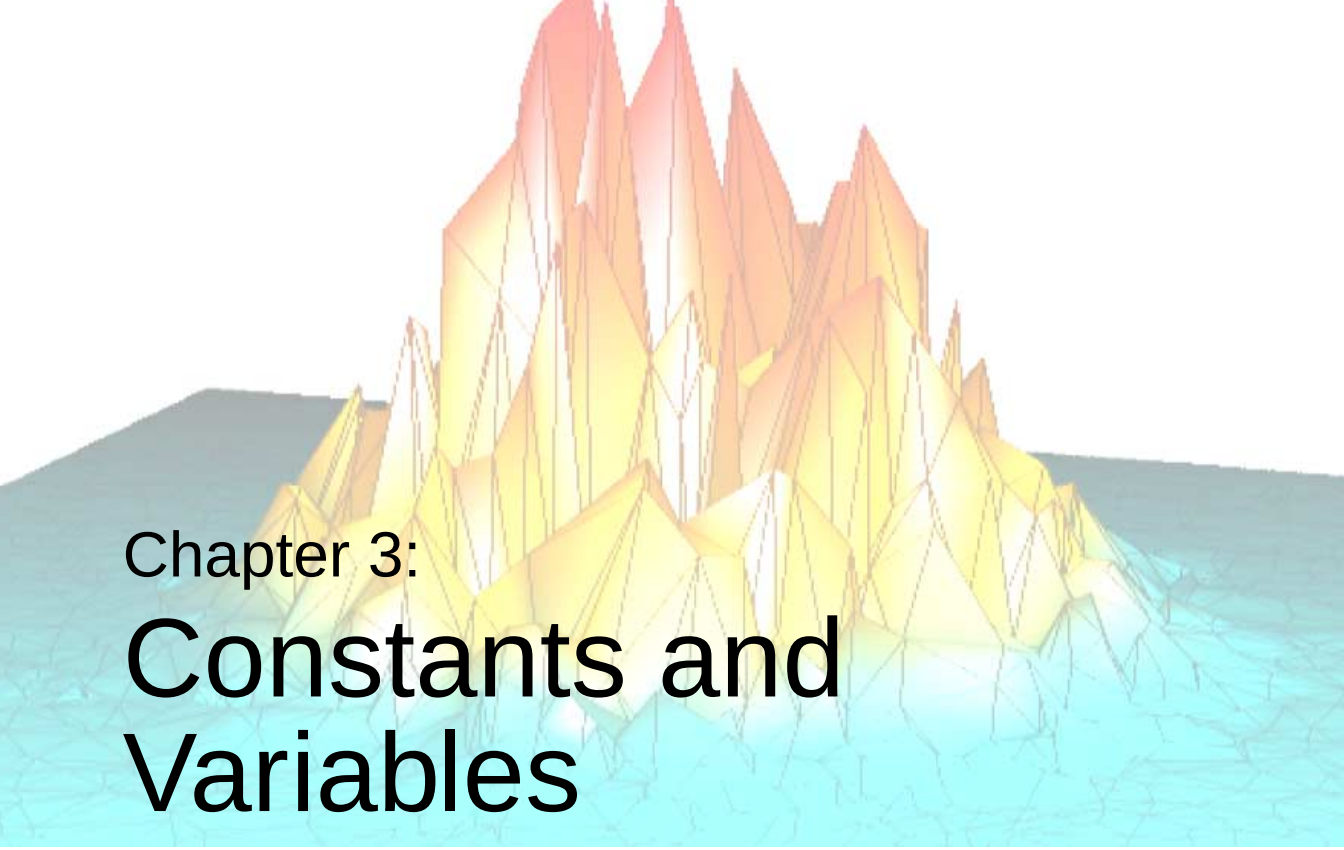
```
;Yields fltarr(4).
FLTARR(4) + FLTARR(1, 4)
```

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.



Chapter 3: Constants and Variables

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.

Basic Data Types

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

IDL's basic data types are discussed in more detail beginning with "Constants" on page 49.

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

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

Note

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

- String: A sequence of characters, from 0 to 2147483647 (2.1 GB) characters in length, which is interpreted as text.

Precision of Floating-Point Numbers

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

For more information on floating-point mathematics, see Chapter 22, "Mathematics" in the *Using IDL* manual. For information on your machine's precision, see "MACHAR" in the *IDL Reference Guide* manual.

Complex Data Types

- Structures: Aggregations of data of various types. Structures are discussed in Chapter 7, "Structures".
- Pointers: A reference to a dynamically-allocated *heap variable*. Pointers are discussed in Chapter 8, "Pointers".
- Object References: A reference to a special heap variable that contains an IDL object structure. Object references are discussed in Chapter 22, "Object Basics".

Determining the Data Type of a Variable or Array

The SIZE function can be used to determine the data type of a variable. See “Determining the Size/Type of an Array” on page 378 for an example.

Constants

Integer Constants

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

Radix	Type	Form	Examples
Decimal	Byte	*n*B	12B, 34B
	Integer	n or nS	12,12S,425,425S
	Unsigned Integer	nU or nUS	12U,12US
	Long	*n*L	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	'0F'X
	Unsigned Integer	'n'XU	’0F’XU
	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 3-1: Integer Constants

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

Table 3-1: (Continued) Integer Constants

Digits in hexadecimal constants include the letters A through F for the decimal numbers 10 through 15. Octal constant use the same style as hexadecimal constants, substituting an O for the X. Absolute values of integer constants are given in the following table.

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

Table 3-2: 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 3-2: (Continued) Absolute Value Range Of Integer Constants

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

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

Table 3-3: Examples of Integer Constants

Floating-Point and Double-Precision Constants

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

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

Form	Example
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 3-4: Syntax of Floating-Point Constants

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

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 $1x10^{45}$ 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 3-5: Examples of Valid String Constants

The following table illustrates invalid string constants. In the last entry of the table, "129" is interpreted as an illegal octal constant. This is because a quote character

followed by a digit from 0 to 7 represents an octal numeric constant, not a string, and the character 9 is an illegal octal digit.

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

Table 3-6: Examples of Invalid String Constants

Note

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.

Representing Non-Printable Characters

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

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

Table 3-7: Specifying Non-Printable Characters

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

Type Conversion Functions

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

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

Table 3-8: Type Conversion Functions

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

Take Care When Converting Types

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

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

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

IDL prints:

```
-32536
```

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

With most floating-point operations, error conditions can be monitored using the FINITE and CHECK_MATH functions. See Chapter 18, “Controlling Errors”, for more information.

Converting Strings

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

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

Dynamic Type Conversion

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

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

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

   ; Prompt the user for a numeric value:
   READ, UserVal, PROMPT='Enter any Numeric Value: '
   ; Convert the user value to the type stored in typeA:
   ConvUserVal = FIX(UserVal, TYPE=typeA)
```

```
   PRINT, ConvUserVal
END
```

Examples of Type Conversion

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

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

Table 3-9: Uses of Type Conversion Functions

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" in the *IDL Reference Guide* manual.

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 3-10: Unacceptable and Acceptable IDL Variable Names

Warning

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

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

AND	BEGIN	BREAK
CASE	COMMON	COMPILE_OPT
CONTINUE	DO	ELSE
END	ENDCASE	ENDELSE
ENDFOR	ENDIF	ENDREP
ENDSWITCH	ENDWHILE	EQ
FOR	FORWARD_FUNCTION	FUNCTION

Table 3-11: IDL Reserved Words

GE	GOTO	GT
IF	INHERITS	LE
LT	MOD	NE
NOT	OF	ON_IOERROR
OR	PRO	REPEAT
SWITCH	THEN	UNTIL
WHILE	XOR	

Table 3-11: (Continued) IDL Reserved Words

System Variables

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

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

System variables are discussed in Appendix D, “System Variables” in the *IDL Reference Guide* manual.

Common Blocks

Common blocks are useful when there are variables that need to be accessed by several IDL procedures or when the value of a variable within a procedure must be preserved across calls. Once a common block has been defined, any program unit referencing that common block can access variables in the block as though they were local variables. Variables in a common statement have a global scope within procedures defining the same common block. Unlike local variables, variables in common blocks are not destroyed when a procedure is exited.

There are two types of common block statements: definition statements and reference statements.

Common Block Definition Statements

The common block definition statement creates a common block with the designated name and places the variables whose names follow into that block. Variables defined in a common block can be referenced by any program unit that declares that common block. The general form of the COMMON block definition statement is as follows:

```
COMMON Block_Name, Variable1, Variable2, ..., Variablen
```

The number of variables appearing in the common block definition statement determines the size of the common block. The first program unit (main program, function, or procedure) defining the common block sets the size of the common block, which can never be expanded. Other program units can reference the common block with any number of variables up to the number originally specified. Different program units can give the variables different names, as shown in the example below.

Common blocks share the same space for all procedures. In IDL, common block variables are matched variable to variable, unlike FORTRAN, where storage locations are matched. The third variable in a given IDL common block will always be the same as the third variable in all declarations of the common block regardless of the size, type, or structure of the preceding variables.

Note that common blocks must appear before any of the variables they define are referenced in the procedure.

Variables in common blocks can be of any type and can be used in the same manner as normal variables. Variables appearing as parameters cannot be used in common blocks. There are no restrictions on the number of common blocks used, although each common block uses dynamic memory.

Example

The two procedures in the following example show how variables defined in common blocks are shared.

```
PRO ADD, A
   COMMON SHARE1, X, Y, Z, Q, R
   A = X + Y + Z + Q + R
   PRINT, X, Y, Z, Q, R, A
   RETURN
END

PRO SUB, T
   COMMON SHARE1, A, B, C, D
   T = A - B - C - D
   PRINT, A, B, C, D, T
   RETURN
END
```

The variables X, Y, Z, and Q in the procedure ADD are the same as the variables A, B, C, and D, respectively, in procedure SUB. The variable R in ADD is not used in SUB. If the procedure SUB were to be compiled before the procedure ADD, an error would occur when the COMMON definition in ADD was compiled. This is because SUB has already declared the size of the common block, SHARE1, which cannot be extended.

Common Block Reference Statements

The common block reference statement duplicates the common block and variable names from a previous definition. The common block need only be defined in the first routine to be compiled that references the block.

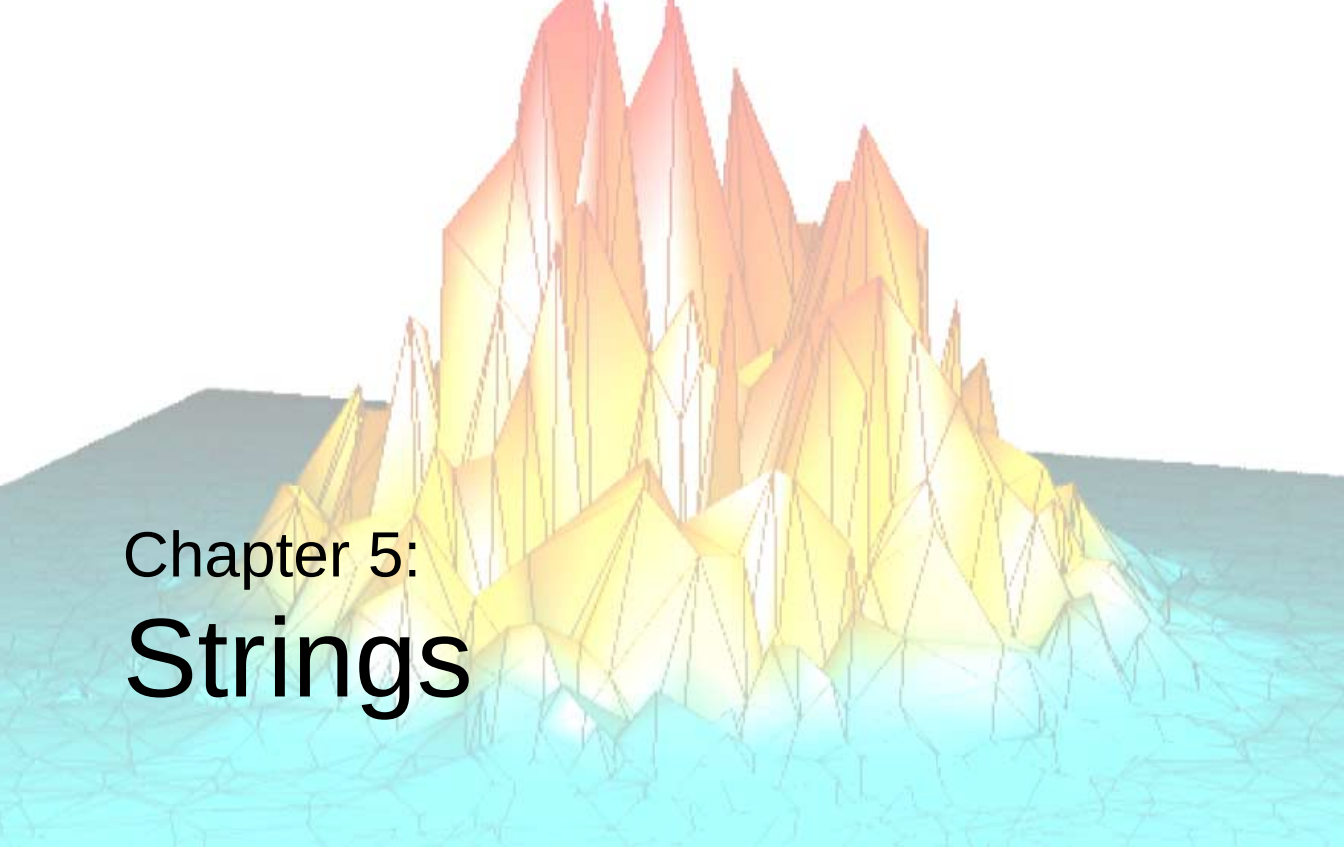
Example

The two procedures in the following example share the common block SHARE2 and all its variables.

```
PRO MULT, M
   COMMON SHARE2, E, F, G
   M = E * F * G
   PRINT, M, E, F, G
   RETURN
END

PRO DIV, D
   COMMON SHARE2
   D = E / F
```

```
    PRINT, D, E, F, G
    RETURN
END
```

The MULT procedure uses a common block *definition* statement to define the block SHARE2. The DIV procedure then uses a common block *reference* statement to gain access to all the variables defined in SHARE2. (Note that MULT must be defined before DIV in order for the common block *reference* to succeed.)



Chapter 5: Strings

The following topics are covered in this chapter:

Overview

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.

A Note About the Examples

In some of the examples in this chapter, it is assumed that a string array named TREES exists. TREES contains the names of seven trees, one name per element, and is created using the statement:

```
trees = ['Beech', 'Birch', 'Mahogany', 'Maple', 'Oak', $
         'Pine', 'Walnut']
```

Executing the statement,

```
PRINT, '>' + trees + '< '
```

results in the following output:

```
>Beech<  >Birch<  >Mahogany<  >Maple<  >Oak<  >Pine<  >Walnut<
```

String Operations

IDL supports several basic string operations, as described below.

Concatenation

The Addition operator, “+”, can be used to concatenate strings together.

Formatting Data

The STRING function is used to format data into a string. The READS procedure can be used to read values from a string into IDL variables.

Case Folding

The STRLOWCASE function returns a copy of its string argument converted to lowercase. Similarly, the STRUPCASE function converts its argument to uppercase.

White Space Removal

The STRCOMPRESS and STRTRIM functions can be used to eliminate unwanted white space (blanks or tabs) from their string arguments.

Length

The STRLEN function returns the length of its string argument.

Substrings

The STRPOS, STRPUT, and STRMID routines locate, insert, and extract substrings from their string arguments.

Splitting and Joining Strings

The STRSPLIT function is used to break strings apart, and the STRJOIN function can be used to and glue strings together.

Comparing Strings

The STRCMP, STRMATCH, and STREGEX functions perform string comparisons.

Non-string and Non-scalar Arguments

Most of the string processing routines described in this chapter expect at least one argument that is the string on which they act.

If the argument is not of type string, IDL converts it to type string using the same default formatting rules that are used by the PRINT/PRINTF or STRING routines. The function then operates on the converted result. Thus, the IDL statement,

```
PRINT, STRLEN(23)
```

returns the result

```
8
```

because the argument “23” is first converted to the string ' 23' that happens to be a string of length 8.

If the argument is an array instead of a scalar, the function returns an array result with the same structure as the argument. Each element of the result corresponds to an element of the argument. For example, the following statements:

```
;Get an uppercase version of TREES.
A = STRUPCASE(trees)

;Show that the result is also an array.
HELP, A

;Display the original.
PRINT, trees

;Display the result.
PRINT, A
```

produce the following output:

```
A               STRING    = Array(7)
Beech Birch Mahogany Maple Oak Pine Walnut
BEECH BIRCH MAHOGANY MAPLE OAK PINE WALNUT
```

For more details on how individual routines handle their arguments, see the individual descriptions in the *IDL Reference Guide.*

String Concatenation

The addition operator is used to concatenate strings. For example, the command:

```
A = 'This is' + ' a concatenation example.'
PRINT, A
```

results in the following output:

```
This is a concatenation example.
```

The following IDL statements build a scalar string containing a comma-separated list of the names found in the TREES string array:

```
;Use REPLICATE to make an array with the correct number of commas
;and add it to trees.
names = trees + [REPLICATE(',', N_ELEMENTS(trees)-1), '']

;Show the resulting list.
PRINT, names
```

Running the above statements results in the following output:

```
Beech, Birch, Mahogany, Maple, Oak, Pine, Walnut
```

Using STRING to Format Data

The STRING function has the following form:

S = 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 221) for details of data formatting. For more information on the STRING function, see "STRING" in the *IDL Reference Guide* manual.

As a simple example, the following IDL statements:

```
;Produce a string array.
A = STRING(FORMAT='("The values are:", /, (I))', INDGEN(5))

;Show its structure.
HELP, A

;Print the result.
FOR I = 0, 4 DO PRINT, A[I]
```

produce the following output:

```
A   STRING    = Array(6)
The values are:
     0
     1
     2
     3
```

Reading Data from Strings

The READS procedure performs formatted input from a string variable and writes the results into one or more output variables. This procedure differs from the READ procedure only in that the input comes from memory instead of a file.

This routine is useful when you need to examine the format of a data file before reading the information it contains. Each line of the file can be read into a string using READF. Then the components of that line can be read into variables using READS.

See the description of "READS" in the *IDL Reference Guide* manual 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 10, “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 56.

Case Folding

The and STRUPCASE functions are used to convert arguments to lowercase or uppercase. They have the form:

S = STRLOWCASE(*String*)

S = STRUPCASE(*String*)

where *String* is the string to be converted to lowercase or uppercase.

The following IDL statements generate a table of the contents of TREES showing each name in its actual case, lowercase and uppercase:

```
FOR I=0, 6 DO PRINT, trees[I], STRLOWCASE(trees[I]),$
STRUPCASE(trees[I]), FORMAT = '(A, T15, A, T30, A)'
```

The resulting output from running this statement is as follows:

Beech	beech	BEECH
Birch	birch	BIRCH
Mahogany	mahogany	MAHOGANY
Maple	maple	MAPLE
Oak	oak	OAK
Pine	pine	PINE
Walnut	walnut	WALNUT

A common use for case folding occurs when writing IDL procedures that require input from the user. By folding the case of the response, it is possible to handle responses written in uppercase, lowercase, or mixed case. For example, the following IDL statements can be used to ask “yes or no” style questions:

```
;Create a string variable to hold the response.
answer = ''

;Ask the question.
READ, 'Answer yes or no:  ', answer
IF (STRUPCASE(answer) EQ 'YES') THEN $
   ;Compare the response to the expected answer.
   PRINT,'YES' ELSE PRINT, 'NO'
```

Whitespace

The STRCOMPRESS and STRTRIM functions are used to remove unwanted white space (tabs and spaces) from a string. This can be useful when reading string data from arbitrarily formatted strings.

Removing All Whitespace

The function STRCOMPRESS returns a copy of its string argument with all white space replaced with a single space or completely removed. It has the form:

S = STRCOMPRESS(*String*)

where *String* is the string to be compressed.

The default action is to replace each section of white space with a single space. Setting the REMOVE_ALL keyword causes white space to be completely eliminated. For example,

```
;Create a string with undesirable white space. Such a string might
;be the result of reading user input with a READ statement.
A = '   This   is  a poorly   spaced  sentence.   '

;Print the result of shrinking all white space to a single blank.
PRINT, '>', STRCOMPRESS(A), '<'

;Print the result of removing all white space.
PRINT '>', STRCOMPRESS(A, /REMOVE_ALL), '<'
```

results in the output:

```
> This is a poorly spaced sentence. <
>Thisisapoorlyspacedsentence.<
```

Removing Leading or Trailing Blanks

The function STRTRIM returns a copy of its string argument with leading and/or trailing white space removed. It has the form:

S = STRTRIM(*String*[, *Flag*])

where *String* is the string to be trimmed and *Flag* is an integer that indicates the specific trimming to be done. If *Flag* is 0 or is not present, trailing white space is removed. If it is 1, leading white space is removed. Both trailing and leading white space are removed if *Flag* is equal to 2. For example:

```
;Create a string with unwanted leading and trailing blanks.
```

```
A = ' This string has leading and trailing white space  '

;Remove trailing white space.
PRINT, '>', STRTRIM(A), '<'

;Remove leading white space.
PRINT, '>', STRTRIM(A,1), '<'

;Remove both.
PRINT, '>', STRTRIM(A,2), '<'
```

Executing these statements produces the output below.

```
>     This string has leading and trailing white space<
>This string has leading and trailing white space    <
>This string has leading and trailing white space<
```

Removing All Types of Whitespace

When processing string data, STRCOMPRESS and STRTRIM can be combined to remove leading and trailing white space and shrink any white space in the middle down to single spaces.

```
;Create a string with undesirable white space.
A =  'Yet   another  poorly  spaced    sentence.   '

;Eliminate unwanted white space.
PRINT, '>' STRCOMPRESS(STRTRIM(A,2)),   '<'
```

Executing these statements gives the result below:

```
>Yet another poorly spaced sentence.<
```

Finding the Length of a String

The STRLEN function is used to obtain the length of a string. It has the form:

L = 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.

```
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

     ;Increment I so as not to count the same instance of 'dog'
     ;twice.
     I = I + 1

   ENDIF
ENDWHILE

;Print the result.
PRINT, 'Found ', cnt, " occurrences of 'dog'"
END
```

Running the above program produces the result below.

```
Found         3 occurrences of 'dog'
```

Searching For the Last Occurrence of a Substring

The REVERSE_SEARCH keyword to the STRPOS function makes it easy to find the last occurrence of a substring within a string. In the following example, we search for the last occurrence of the letter “I” (or “i”) in a sentence:

```
sentence = 'IDL is fun.'
sentence = STRUPCASE(sentence)
lasti = STRPOS(sentence, 'I', /REVERSE_SEARCH)
PRINT, lasti
```

This results in:

```
4
```

Note that although REVERSE_SEARCH tells STRPOS to begin searching from the end of the string, the STRPOS function still returns the position of the search string starting from the beginning of the string (where 0 is the position of the first character).

Inserting the Contents of One String into Another

The STRPUT procedure is used to insert the contents of one string into another. It has the form,

STRPUT, *Destination*, *Source*[, *Position*]

where *Destination* is the string to be overwritten, *Source* is the string to be inserted, and *Position* is the first character position within *Destination* at which *Source* will be inserted. If the optional argument *Position* is omitted, the overwrite is started at the first character (character position 0). The following IDL statements use STRPOS and STRPUT to replace every occurrence of the word “dog” with the word “CAT” in the string “dog cat duck rabbit dog cat dog”:

```
animals = 'dog cat duck rabbit dog cat dog'
;The string to search, "dog", appears three times.

;While any occurrence of "dog" exists, replace it.
WHILE (((I = STRPOS(animals, 'dog'))) NE -1) DO $
STRPUT, animals, 'CAT', I

;Show the resulting string.
PRINT, animals
```

Running the above statements produces the result below.

```
CAT cat duck rabbit CAT cat CAT
```

Extracting Substrings

The STRMID function is used for extracting substrings from a larger string. It has the form:

STRMID(*Expression*, *First_Character* [, *Length*])

where *Expression* is the string from which the substring will be extracted, *First_Character* is the starting position within *Expression* of the substring (the first position is position 0), and *Length* is the length of the substring to extract. If there are not *Length* characters following the position *First_Character*, the substring will be truncated. If the *Length* argument is not supplied, STRMID extracts all characters from the specified starting position to the end of the string. The following IDL statements use STRMID to print a table matching the number of each month with its three-letter abbreviation:

```
;String containing all the month names.
months = 'JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC'

;Extract each name in turn. The equation (I-1)*3 calculates the
;position within MONTH for each abbreviation
FOR I = 1, 12 DO PRINT, I, '      ', $
STRMID(months, (I - 1) * 3, 3)
```

The result of executing these statements is as follows:

```
 1      JAN
 2      FEB
 3      MAR
 4      APR
 5      MAY
 6      JUN
 7      JUL
 8      AUG
 9      SEP
10      OCT
11      NOV
12      DEC
```

Splitting and Joining Strings

The STRSPLIT function is used to break apart a string, and the STRJOIN function is used to glue together separate strings into a single string.

The STRSPLIT function uses the following syntax:

Result = STRSPLIT(*String* [, *Pattern*])

where *String* is the string to be split, and *Pattern* is either a string of character codes used to specify the delimiter, or a regular expression, as implemented by the STREGEX function.

The STRJOIN function uses the following syntax:

Result = STRJOIN(*String* [, *Delimiter*])

where *String* is the string or string array to be joined, and *Delimiter* is the separator string to use between the joined strings.

The following example uses STRSPLIT to extract words from a sentence into an array, modifies the array, and uses STRJOIN to rejoin the individual array elements into a new sentence:

```
str1 = 'Hello Cruel World'
words = STRSPLIT(str1, ' ', /EXTRACT)
newwords=[words[0],words[2]]
PRINT, STRJOIN(newwords, ' ')
```

This code results in the following output:

```
Hello World
```

In this example, the EXTRACT keyword caused STRSPLIT to return the substrings as array elements, rather than the default action of returning an array of character offsets indicating the position of each substring.

The STRJOIN function allows us to specify the delimiter used to join the strings. Instead of using a space as in the above example, we could use a different delimiter as follows:

```
str1 = 'Hello Cruel World'
words = STRSPLIT(str1, ' ', /EXTRACT)
newwords=[words[0],words[2]]
PRINT, STRJOIN(newwords, ' Kind ')
```

This code results in the following output:

```
Hello Kind World
```

Comparing Strings

IDL provides several different mechanisms for performing string comparisons. In addition to the EQ operator, the STRCMP, STRMATCH, and STREGEX functions can all be used for string comparisons.

Case-Insensitive Comparisons of the First N Characters

The STRCMP function simplifies case-insensitive comparisons, and comparisons of only the first N characters of two strings. The STRCMP function uses the following syntax:

Result = STRCMP(*String1*, *String2* [, *N*])

where *String1* and *String2* are the strings to be compared, and *N* is the number of characters from the beginning of the string to compare.

Using the EQ operator to compare the first 3 characters of the strings “Moose” and “mOO” requires the following steps:

```
A = 'Moose'
B = 'mOO'

C=STRMID(A,0,3)

IF (STRLOWCASE(C) EQ STRLOWCASE(B)) THEN PRINT, "It's a match!"
```

Using the EQ operator for this case-insensitive comparison of the first 3 characters requires the STRMID function to extract the first 3 characters, and the STRLOWCASE (or STRUPCASE) function to change the case.

The STRCMP function could be used to simplify this comparison:

```
A='Moose'
B='mOO'

IF (STRCMP(A,B,3, /FOLD_CASE) EQ 1) THEN PRINT, "It's a match!"
```

The optional *N* argument of the STRCMP function allows us to easily specify how many characters to compare (from the beginning of the input strings), and the FOLD_CASE keyword specifies a case-insensitive search. If *N* is omitted, the full strings are compared.

String Comparisons Using Wildcards

The STRMATCH function can be used to compare a search string containing wildcard characters to another string. It is similar in function to the way the standard UNIX command shell processes file wildcard characters.

The STRMATCH function uses the following syntax:

Result = STRMATCH(*String*, *SearchString*)

where *String* is the string in which to search for *SearchString*.

SearchString can contain the following wildcard characters:

Wildcard Character	Description
*	Matches any string, including the null string.
?	Matches any single character.
[...]	Matches any one of the enclosed characters. A pair of characters separated by "-" matches any character lexically between the pair, inclusive. If the first character following the opening [is a !, any character not enclosed is matched. To prevent one of these characters from acting as a wildcard, it can be quoted by preceding it with a backslash character (e.g. "*" matches the asterisk character). Quoting any other character (including \ itself) is equivalent to the character (e.g. "\a" is the same as "a").

Table 5-1: Wildcard Characters used by STRMATCH

The following examples demonstrate various uses of wildcard matching:

Example 1: Find all 4-letter words in a string array that begin with “f” or “F” and end with “t” or “T”:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f??t', /FOLD_CASE) EQ 1)]
```

This results in:

```
foot Feet FAST fort
```

Example 2: Find words of any length that begin with “f” and end with “t”:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
```

```
PRINT, str[WHERE(STRMATCH(str, 'f*t', /FOLD_CASE) EQ 1)]
```

This results in:

```
foot Feet FAST ferret fort
```

Example 3: Find 4-letter words beginning with “f” and ending with “t”, with any combination of “o” and “e” in between:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f[eo][eo]t', /FOLD_CASE) EQ 1)]
```

This results in:

```
foot Feet
```

Example 4: Find all words beginning with “f” and ending with “t” whose second character is not the letter “o”:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f[!o]*t', /FOLD_CASE) EQ 1)]
```

This results in:

```
Feet FAST ferret
```

Complex Comparisons Using Regular Expressions

A more difficult search than the one above would be to find words of any length beginning with “f” and ending with “t” without the letter “o” in between. This would be difficult to accomplish with STRMATCH, but could be easily accomplished using the STREGEX function:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, STREGEX(str, '^f[^o]*t$', /EXTRACT, /FOLD_CASE)
```

This statement results in:

```
Feet FAST ferret
```

Note the following about this example:

- Unlike the * wildcard character used by STRMATCH, the * meta character used by STREGEX applies to the item directly on its left, which in this case is [^o], meaning “any character except the letter ‘o’ ”. Therefore, [^o]* means “zero or more characters that are not ‘o’ ”, whereas the following statement would find only words whose second character is not “o”:

```
PRINT, str[WHERE(STRMATCH(str, 'f[!o]*t', /FOLD_CASE) EQ 1)]
```

- The anchors (^ and $) tell STREGEX to find only words that begin with “f” and end with “t”. If we left out the $ anchor, STREGEX would also return “fat”, which is a substring of “fate”.

Regular expressions are somewhat more difficult to use than simple wildcard matching (which is why the UNIX shell does matching) but in exchange offers unparalleled expressive power.

For more on the STREGEX function, see “STREGEX” in the *IDL Reference Guide* manual, and for an introduction to regular expressions, see “Learning About Regular Expressions” on page 117.

Learning About Regular Expressions

Regular expressions are a very powerful way to match arbitrary text. Stemming from neurophysiological research conducted in the early 1940's, their mathematical foundation was established during the 1950's and 1960's. Their use has a long history in computer science, and they are an integral part of many UNIX tools, including awk, egrep, lex, perl, and sed, as well as many text editors. Regular expressions are slower than simple pattern matching algorithms, and they can be cryptic and difficult to write correctly. Small mistakes in specification can yield surprising results. They are, however, vastly more succinct and powerful than simple pattern matching, and can easily handle tasks that would be difficult or impossible otherwise.

The topic of regular expressions is a very large one, complicated by the arbitrary differences in the implementations found in various tools. Anything beyond an extremely simplistic sketch is well beyond the scope of this manual. To understand them better, we recommend a good text on the subject, such as "Mastering Regular Expressions", by Jeffrey E.F. Friedl (O'Reilly & Associates, Inc, ISBN 1-56592-257-3). The following is an abbreviated, simplified, and incomplete explanation of regular expressions, sufficient to gain a cursory understanding of them.

The regular expression engine attempts to match the regular expression against the input string. Such matching starts at the beginning of the string and moves from left to right. The matching is considered to be "greedy", because at any given point, it will always match the longest possible substring. For example, if a regular expression could match the substring 'aa' or 'aaa', it will always take the longer option.

Meta Characters

A regular expression "ordinary character" is a character that matches itself. Most characters are ordinary. The exceptions, sometimes called "meta characters", have special meanings. To convert a meta character into an ordinary one, you "escape" it by preceding it with a backslash character (e.g. '*'). The meta characters are described in the following table:

Character	Description
.	The period matches any character.

Table 5-2: Meta characters

Character	Description
[ ]	The open bracket character indicates a “bracket expression”, which is discussed below. The close bracket character terminates such an expression.
\	The backslash suppresses the special meaning of the character it precedes, and turns it into an ordinary character. To insert a backslash into your regular expression pattern, use a double backslash ('\\').
( )	The open parenthesis indicates a “subexpression”, discussed below. The close parenthesis character terminates such a subexpression.
Repetition Characters	These characters are used to specify repetition. The repetition is applied to the character or expression directly to the left of the repetition operator.
*	Zero or more of the character or expression to the left. Hence, 'a*' means “zero or more instances of 'a' ”.
+	One or more of the character or expression to the left. Hence, 'a+' means “one or more instances of 'a'”.
?	Zero or one of the character or expression to the left. Hence, 'a?' will match 'a' or the empty string ''.
{}	An interval qualifier allows you to specify exactly how many instances of the character or expression to the left to match. If it encloses a single unsigned integer length, it means to match exactly that number of instances. Hence, 'a{3}' will match 'aaa'. If it encloses 2 such integers separated by a comma, it specifies a range of possible repetitions. For example, 'a{2,4}' will match 'aa', 'aaa', or 'aaaa'. Note that '{0,1}' is equivalent to '?'.
|	Alternation. This operator is used to indicate that one of several possible choices can match. For example, '(a|b|c)z' will match any of 'az', 'bz', or 'cz'.

Table 5-2: (Continued) Meta characters

Character	Description
^ $	Anchors. A '^' matches the beginning of a string, and '$' matches the end. As we have seen above, regular expressions usually match any possible substring. Anchors can be used to change this and require a match to occur at the beginning or end of the string. For example, '^abc' will only match strings that start with the string 'abc'. '^abc$' will only match a string containing *only* 'abc'.

Table 5-2: (Continued) Meta characters

Subexpressions

Subexpressions are those parts of a regular expression enclosed in parentheses. There are two reasons to use subexpressions:

- To apply a repetition operator to more than one character. For example, '(fun){3}' matches 'funfunfun', while 'fun{3}' matches 'funnn'.
- To allow location of the subexpression using the SUBEXPR keyword to STREGEX.

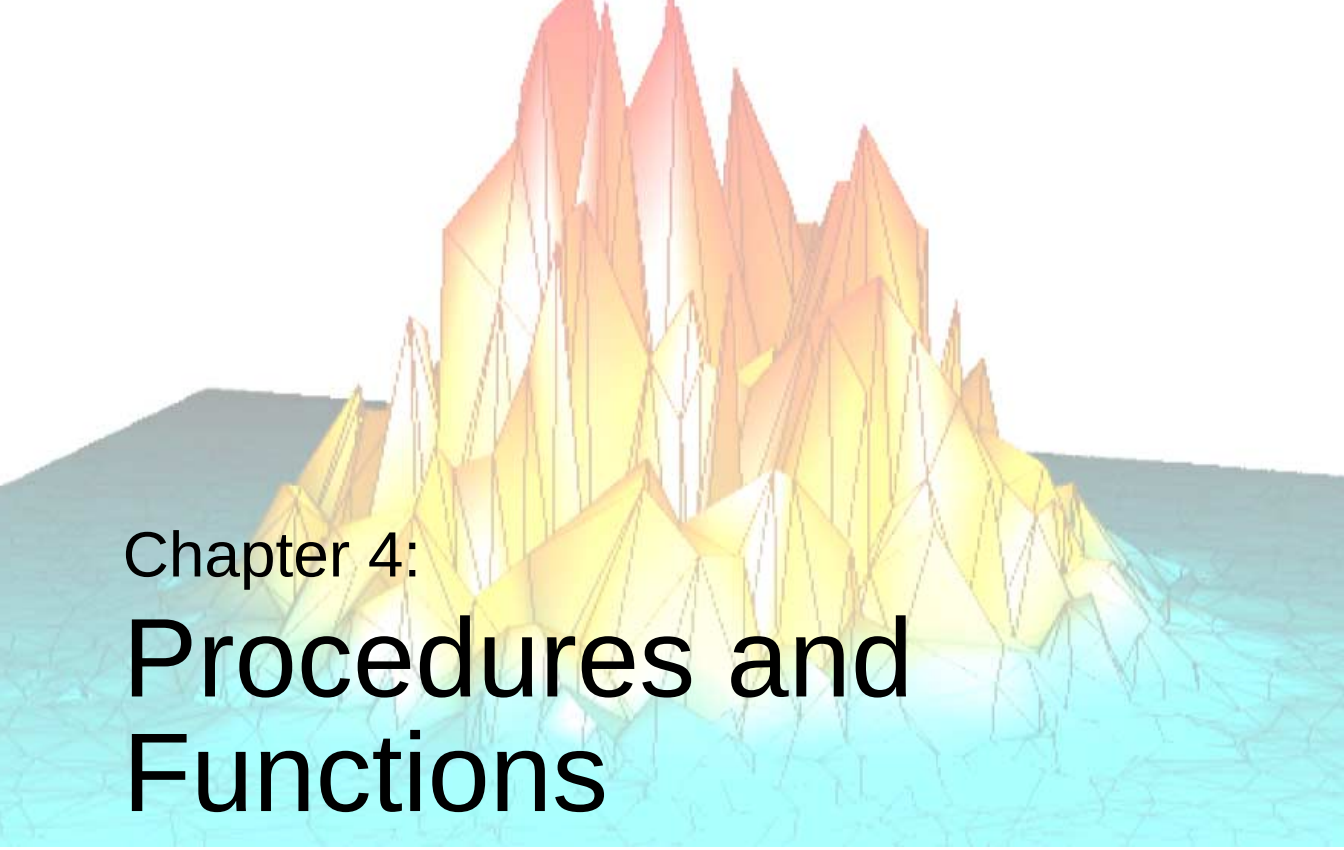
Bracket Expressions

Bracket expressions (expressions enclosed in square brackets) are used to specify a set of characters that can satisfy a match. Many of the meta characters described above (.*[\) lose their special meaning within a bracket expression. The right bracket loses its special meaning if it occurs as the first character in the expression (after an initial '^', if any).

There are several different forms of bracket expressions, including:

- **Matching List** — A matching list expression specifies a list that matches any one of the characters in the list. For example, '[abc]' matches any of the characters 'a', 'b', or 'c'.
- **Non-Matching List** — A non-matching list expression begins with a '^', and specifies a list that matches any character *not* in the list. For example, '[^abc]' matches any characters *except* 'a', 'b', or 'c'. The '^' only has this special meaning when it occurs first in the list immediately after the opening '['.
- **Range Expression** — A range expression consists of 2 characters separated by a hyphen, and matches any characters lexically within the range indicated. For

example, '[A-Za-z]' will match any alphabetic character, upper or lower case. Another way to get this effect is to specify '[a-z]' and use the FOLD_CASE keyword to STREGEX.



Chapter 4: Procedures and Functions

The following topics are covered in this chapter:

Overview

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

New procedures and functions can be written in IDL and called in the same manner as the system-defined procedures or functions from the keyboard or from other programs. When a procedure or function is finished, it executes a RETURN statement that returns control to its caller. Functions always return an explicit result. A procedure is called by a procedure call statement, while a function is called by a function reference. For example, if ABC is a procedure and XYZ is a function, the calling syntax is:

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

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

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 "Defining a Function" on page 71.
- Procedures written in IDL, that are included with the IDL distribution, located in directories that are specified in the search path.
- Under many operating systems, user-written system procedures coded in FORTRAN, C, or any language that follows the standard calling conventions, which have been dynamically linked with IDL using the LINKIMAGE or CALL_EXTERNAL procedures.

Example

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

```
ERASE
```

This is a procedure call to a subroutine to erase the screen. There are no explicit inputs or outputs. Other procedures have one or more parameters. For example, the statement:

```
PLOT, CIRCLE
```

calls the PLOT procedure with the parameter CIRCLE.

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. Once a function has been defined, references to the function cause the program unit to be executed. All functions return a function value which is given as a parameter in the RETURN statement used to exit the function. Function names can be up to 128 characters long.

The general format of a function definition is as follows:

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

Example

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

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

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

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

squares the array X, passes this result to the AVERAGE function, and prints the result. Parameters passed to functions are identified by their position or by a keyword. See “Using Keyword Parameters” on page 77.

Automatic Execution

IDL automatically compiles and executes a user-written function or procedure when it is first referenced if:

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

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

Note

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

Warning

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

For information on how to access routines, see "Running IDL Program Files" in Chapter 9 of the *Using IDL* manual.

Forward Function Definition

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

This problem has been addressed beginning with IDL version 5.0 by the use of square brackets "[ ]" instead of parentheses to specify array subscripts. See "Array Subscript Syntax: [ ] vs. ( )" on page 128 for a discussion of the IDL version 5.0 and later syntax. However, because parentheses are still allowed in array subscripting statements, the need for a mechanism by which the programmer can "reserve" a name for a function that has not yet been defined remains. The FORWARD_FUNCTION statement addresses this need.

As mentioned above, ambiguities can arise between function calls and array references when a function has not yet been compiled, or there is no file with the same name as the function found in the IDL path.

For example, attempting to compile the IDL statement:

```
A = xyz(1, COLOR=1)
```

will cause an error if the user-written function XYZ has not been compiled and the filename xyz.pro is not found in the IDL path. IDL reports a syntax error, because xyz is interpreted as an array variable instead of a function name.

This problem can be eliminated by using the FORWARD_FUNCTION statement. This statement has the following syntax:

```
FORWARD_FUNCTION Name1, Name2, ..., NameN
```

where *Name* is the name of a function that has not yet been compiled. Any names declared as forward-defined functions will be interpreted as functions (instead of as variable names) for the duration of the IDL session.

For example, we can resolve the ambiguity in the previous example by adding a FORWARD_FUNCTION definition:

```
;Define XYZ as the name of a function that has not yet been
;compiled.
FORWARD_FUNCTION XYZ

;IDL now understands this statement to be a function call instead
;of a bad variable reference.
a = XYZ(1, COLOR=1)
```

Note

Declaring a function that will be merged into IDL via the LINKIMAGE command with the FORWARD_FUNCTION statement will not have the desired effect. Routines merged via LINKIMAGE are considered by IDL to be built-in routines, and thus need no compilation or declaration. They must, however, be merged with IDL before any routines that call them are compiled.

Parameters

The variables and expressions passed to the function or procedure from its caller are *parameters*. *Actual parameters* are those appearing in the procedure call statement or the function reference. In the examples at the beginning of this section, the actual parameters in the procedure call are the variable A and the constant 12, while the actual parameter in the function call is the value of the expression (C/D).

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

Correspondence of Formal and Actual Parameters

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

Positional Parameters

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

Keyword Parameters

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

When calling a routine with a keyword parameter, you can abbreviate the keyword to its shortest, unambiguous abbreviation. Keyword parameters can also be specified by the caller with the syntax /KEYWORD, which is equivalent to setting the keyword parameter to 1 (e.g., KEYWORD = 1). The syntax /KEYWORD is often referred to, in the rest of this documentation, as *setting* the keyword.

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

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

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

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

```
XYZ, TEST = A

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

;Variable Q is copied to formal parameter A. B and T are undefined
;inside the procedure.
XYZ, Q
```

Note

When supplying keyword parameters for a function, the keyword is specified *inside* the parentheses:

```
result = FUNCTION(Arg1, Arg2, KEYWORD = value)
```

Copying Parameters

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

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

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

Number of Parameters

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

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

Example

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

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

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

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

;End of function.
END
```

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

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

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

;Display gradient of IMAGE sum.
TVSCL, GRAD(abc + def)
```

Using Keyword Parameters

A short example of a function that exchanges two columns of a 4 × 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.

This function example uses the system function KEYWORD_SET to determine if a keyword parameter has been passed and if it is nonzero. This is similar to using the condition:

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

to test if keywords that have a true/false value are both present and true.

Keyword Inheritance

Keyword inheritance allows IDL routines to accept keyword parameters not defined in their function or procedure declaration and pass them on to 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 80. Advice on how to choose the best one for your needs can be found in "Choosing a Keyword Inheritance Mechanism" on page 82. 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 RSI recommends _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 TEST 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:
    $MAIN$  TEST1
Compiled Functions:
```

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 [File1 , Filen, ... ]
.COMPILE [File1 , Filen, ... ]
```

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

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

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

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

Note Regarding Functions

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

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

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

Note

In versions of IDL prior to version 5.0, parentheses were used to enclose array subscripts. While using parentheses to enclose array subscripts will continue to work as in previous version of IDL, we strongly suggest that you use brackets in all

new code. See “Array Subscript Syntax: [ ] vs. ( )” on page 128 for additional details.

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

There are several ways to avoid this problem:

- Compile the lowest-level functions (those that call no other functions) first, then higher-level functions, and finally procedures.
- Place the function in a file with the same name as the function, and place that file in one of the directories specified by !PATH.
- Use the FORWARD_FUNCTION definition statement to inform IDL that a given name refers to a function rather than a variable. See “Forward Function Definition” on page 72.
- Manually compile all functions before any reference, or use RESOLVE_ROUTINE or RESOLVE_ALL to compile functions.

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 6, “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 151 for additional details.

Calling Mechanism

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

1. All of the actual arguments in the user-procedure call list are evaluated and saved in temporary locations.
2. The actual parameters that were saved are substituted for the formal parameters given in the definition of the called procedure. All other variables local to the called procedure are set to undefined.
3. The function or procedure is executed until a RETURN or RETALL statement is encountered. Procedures also can return on an END statement. The result of a user-written function is passed back to the caller by specifying it as the value of a RETURN statement. RETURN statements in procedures cannot specify a return value.
4. All local variables in the procedure, those variables that are neither parameters nor common variables, are deleted.
5. The new values of the parameters that were passed by reference are copied back into the corresponding variables. Actual parameters that were passed by value are deleted.
6. Control resumes in the calling procedure after the procedure call statement or function reference.

Recursion

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

Example

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

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

   ;Save previous record number.
```

```
    COMMON NXT_COM, lastrec

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

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

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

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

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

    RETURN A

END
```

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

Setting Compilation Options

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

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

where opt_n is any of the following:

- **IDL2** — A shorthand way of saying:

  ```
  COMPILE_OPT DEFINT32, STRICTARR
  ```

- **DEFINT32** — IDL should assume that lexical integer constants are the 32-bit LONG type rather than the default of 16-bit integers. This takes effect from the point where the COMPILE_OPT statement appears in the routine being compiled.

- **HIDDEN** — This routine should not be displayed by HELP, unless the FULL keyword to HELP is used. This directive can be used to hide helper routines that regular IDL users are not interested in seeing.

  A side effect of making a routine hidden is that IDL will not print a “Compile module” message for it when it is compiled from the library to satisfy a call to it. This makes hidden routines appear built in to the user.

- **OBSOLETE** — If the user has !WARN.OBS_ROUTINES set to True, attempts to compile a call to this routine will generate warning messages that this routine is obsolete. This directive can be used to warn people that there may be better ways to perform the desired task.

- **STRICTARR** — While compiling this routine, IDL will not allow the use of parenthesis to index arrays, reserving their use only for functions. Square brackets are then the only way to index arrays. Use of this directive will prevent the addition of a new function in future versions of IDL, or new libraries of IDL code from any source, from changing the meaning of your code, and is an especially good idea for library functions.

  Use of STRICTARR can eliminate many uses of the FORWARD_FUNCTION definition.

RSI recommends the use of

```
COMPILE_OPT IDL2
```

in all new code intended for use in a reusable library. We further recommend the use of

```
COMPILE_OPT idl2, HIDDEN
```

in all such routines that are not intended to be called directly by regular users (e.g. helper routines that are part of a larger package).

Advice for Library Authors

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

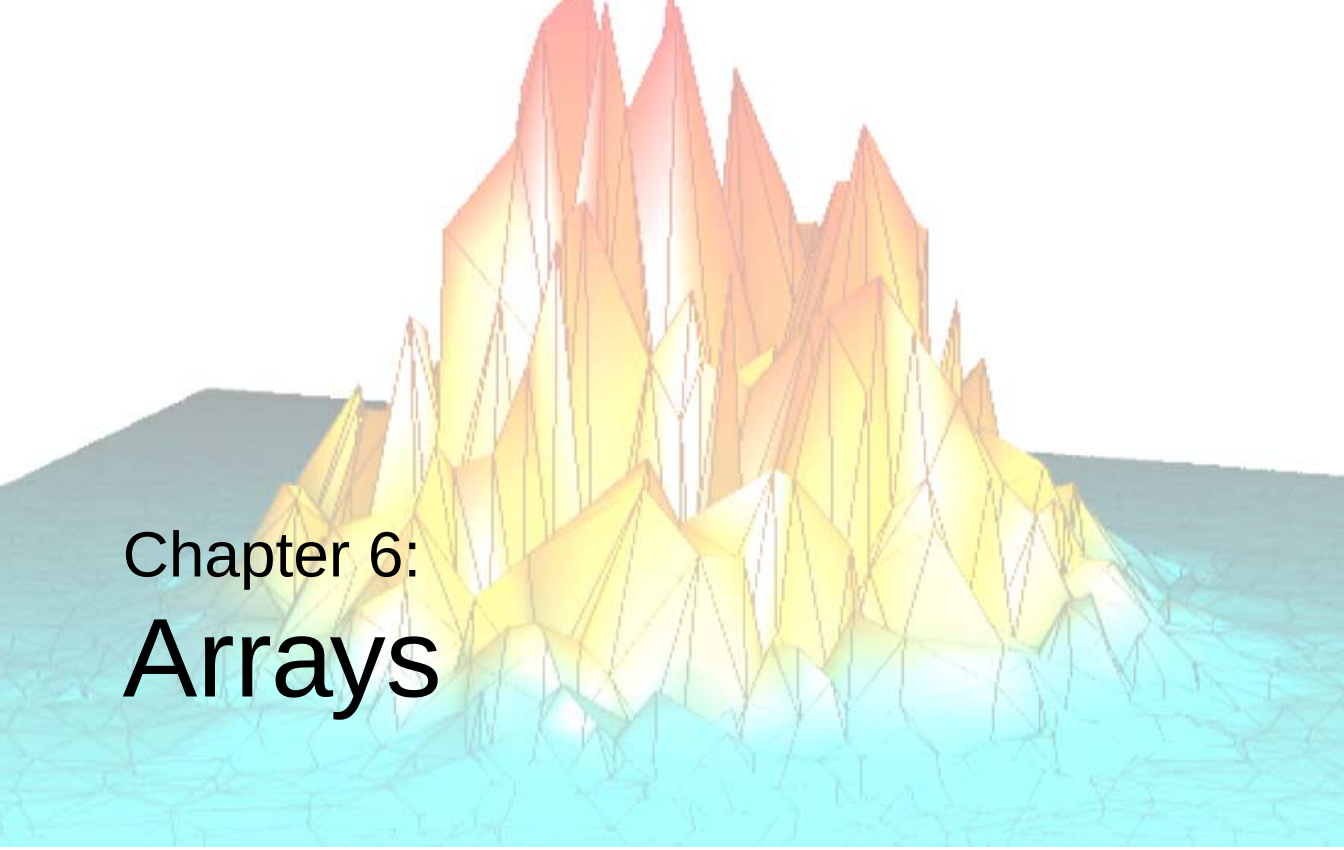
- The structure and organization of the library needs to encourage reuse and generality.
- Library code must be more robust than the usual program. Stability of implementation, and especially of interface, are very important.
- Errors must be gracefully handled whenever possible. See Chapter 18, “Controlling Errors” 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. They must not to alter the global environment in ways that cause conflicts. In doing this, 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 given library from conflicting with other libraries on the same system, and protects the library from changes to IDL that may occur in newer releases.

The need to use a unique prefix for the names in your library is very important. New releases of IDL occur on a regular schedule. These new releases contain new routines, system variables, common blocks, and other globally visible items. If one of these new names is the same as a name used in your library, the conflict will prevent your library from being usable with that new version until you take steps to change the troublesome name. This is difficult for you and inconvenient for your users. The use of a proper prefix eliminates this risk and makes it easier for your library to work with new versions of IDL without the need to take special action with each new IDL release.

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 by RSI. New names of these forms can and will appear without warning in new versions of IDL, and

should be avoided when naming new libraries. Failure to use prefixed naming can lead to considerable difficulty once the library is established. It is important to establish a naming convention early and enforce its systematic use throughout.



Chapter 6: Arrays

The following topics are covered in this chapter:

Overview

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
```

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.

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
```

When a subscript reference appears on the left side of an assignment statement, new values are stored in selected array elements, without altering the remaining elements. For example, the following statements change the third element of array:

```
array = [1.0, 2.0, 3.0, 4.0, 5.0]
array[2] = 9.0
PRINT, array
```

IDL prints:

```
1.00000     2.00000     9.00000     4.00000     5.00000
```

Chapter 11, “Assignment” discusses the use of the different types of assignment statements when storing into arrays.

Similarly, 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
```

Array Subscripting

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. When arrays are used for mathematical operations (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
$A_{0,1}$	$A_{1,1}$	. . .
$A_{0,2}$	$A_{1,2}$	Highest memory address

Table 6-1: 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 138. For a discussion of how the ordering of such data relates to IDL mathematics routines, see "Arrays and Matrices" in Chapter 22 of the *Using IDL* manual.

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 138 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.

Elements of multidimensional arrays also can be specified using only one subscript, in which case the array is treated as a vector with the same number of points. In the above example, `A[2]` is the same element as `A[0, 1]`, and `A[5]` is the same element as `A[1, 2]`.

If an attempt is made to reference a nonexistent element of an array using a scalar subscript (a subscript that is negative or larger than the size of the dimension minus 1), an error occurs and program execution stops.

Subscripts can be any type of vector or scalar expression. If a subscript expression is not integer, a longword integer copy is made and used to evaluate the subscript.

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)
```

Subscripting Scalars

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.”

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.

Subscript Ranges

Subscript ranges are used to select a subarray from an array by giving the starting and ending subscripts of the subarray in each dimension. Subscript ranges can be combined with scalar and array subscripts and with other subscript ranges. Any rectangular portion of an array can be selected with subscript ranges. There are six types of subscript ranges:

- A range of subscripts, written [e_0:e_1], denoting all elements whose subscripts range from the expression e_0 through e_1 (e_0 must not be greater than e_1). For example, if the variable `vec` is a 50-element vector, `vec[5:9]` is a five-element vector composed of `vec[5]` through `vec[9]`.
- A range of subscripts, written [e_0:e_1:e_2], denoting every e_2th element within the range of subscripts *e0* 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]`.
- All elements from a given element to the last element of the dimension, written as [e_0:*]. Using the above example, `vec[10:*]` is a 40-element vector made from `vec[10]` through `vec[49]`.
- Every e_2th element from a given element to the last element of the dimension, written as [e_0:*:e_2]. 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. Using the above example, `vec[10:*:4]` is a 10-element vector made from every fourth element between `vec[10]` through `vec[49]`.
- A simple subscript, [*n*]. When used with multidimensional arrays, simple subscripts specify only elements with subscripts equal to the given subscript in that dimension.
- All elements of a dimension, written [*]. This form is used with multidimensional arrays to select all elements along the dimension. For example, if `arr` is a 10-column by 12-row array, `arr[*, 11]` is the last row of `arr`, composed of elements `[arr[0,11], arr[1,11], ..., arr[9,11]]`, and is a 10-element row vector. Similarly, `arr[0, *]` is the first column of `arr`, `[arr[0,0], arr[0,1],..., arr[0,11]]`, and its dimensions are 1 column by 12 rows.

Multidimensional subarrays can be specified using any combination of the above forms. For example, `arr[*, 0:4]` is made from all columns of rows 0 to 4 of `arr` or a 10-column, 5-row array. The table below summarizes the possible forms of subscript ranges:

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

Table 6-2: Subscript Ranges

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,*]
```

Examples

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,*]
```

See Chapter 11, “Assignment” for a description of the process of assigning values to subarrays.

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 135), 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:

$$V[S] = \begin{cases} V_{S_i} & \text{if} \quad 0 \le S_i < n \\ V_0 & \text{if} \quad S_i < 0 \\ V_{n-1} & \text{if} \quad S_i \ge n \end{cases} \qquad \text{for } 0 \le i < m$$

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*.

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. 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” in the *IDL Reference Guide* manual 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.

Example

As an example, consider the commands:

```
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.

As another example, 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.

The WHERE function, which returns a vector of subscripts, can be used to select elements of an array using expressions similar to `A[WHERE(A GT 0)]`, which results in a vector composed only of the elements of `A` that are greater than 0.

Combining 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, 3×3 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 1×3-column vector `A[8,1]`, `A[8,3]`, and `A[8,5]`, illustrating that leading dimensions are not eliminated.

Storing Elements with Array Subscripts

One or more values can be stored in selected elements of an array by using an array expression as a subscript for the array on the left side of an assignment statement. Values are taken from the expression on the right side of the assignment statement and stored in the elements whose subscripts are given by the array subscript. The right-hand expression can be either a scalar or array.

The subscript array is converted to longword type before use if necessary. Regardless of structure, this subscript array is interpreted as a vector. For details and examples of storing with vector subscripts, see Chapter 11, “Assignment”.

Examples

The statement:

```
A[[2, 4, 6]] = 0
```

zeroes elements `A[2]`, `A[4]`, and `A[6]`, without changing other elements of `A`. The statement:

```
A[[2, 4, 6]] = [4, 16, 36]
```

stores 4 in `A[2]`, 16 in `A[4]`, and 36 in `A[6]`.

One way to create a square $n \times n$ identity matrix is as follows:

```
A = FLTARR(N, N)
A[INDGEN(N) * (N + 1)] = 1.0
```

The expression `INDGEN(N)*(N + 1)` results in a vector containing the subscripts of the diagonal elements `[0, N+1, 2N+2, ..., (N-1)*(N+1)]`.

Yet another way is to use two array subscripts. The statements:

```
A = FLTARR(N, N)
A[INDGEN(N), INDGEN(N)] = 1.0
```

create the array subscripts `[[0,0], [1,1], ..., [n-1, n-1]]`. The statement:

```
A[WHERE(A LE 0)] = -1
```

sets elements of `A` with values of zero or less to -1.

The following statements create a 10x10 identity matrix:

```
A = FLTARR(10, 10)
A[INDGEN(10) * 11] = 1
```

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 6-1, 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 6-1: 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 6-2. 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 6-1, 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 6-2: 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 6-2) 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:

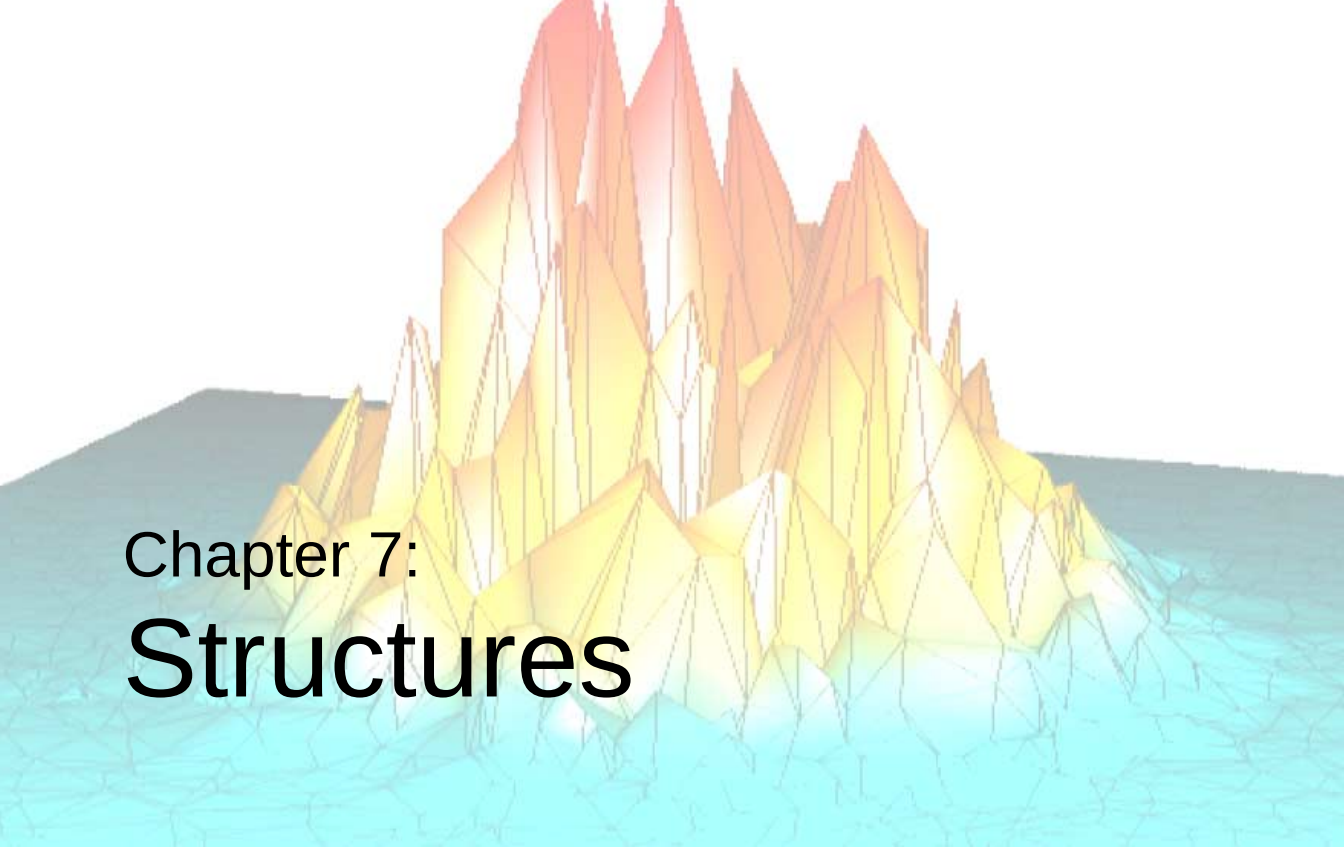
Sharing Data With Other Languages — If binary data written by a row major language is to be input and used by IDL, transposition of the data is usually required first. Similarly, if IDL is writing binary data for use by a program written in a row major language, transposition of the data before writing (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 "Arrays and Matrices" in Chapter 22 of the *Using IDL* manual.

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 7: Structures

The following topics are covered in this chapter:

Overview

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

Named Structures

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

Anonymous Structures

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

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

Creating and Defining Structures

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

{*Structure_Name, Tag_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 159. 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 22, “Object Basics”.

Example of Creating a Structure

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

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

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

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

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

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

or by using the CREATE_STRUCT function:

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

Structure References

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

Variable_Name.Tag_Name

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

```
var.tag1.tag2
```

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 151 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 149 and assuming the STAR structure has been defined, an array containing 100 elements of the structure is created with the following statement:

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

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

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

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

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

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

Examples of Arrays of Structures

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

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

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

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

;Prints name field of all 100 elements of cat, separated by commas
```

```
;(the last field has a trailing comma).
PRINT, cat.name + ','

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

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

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

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

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

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

Structure Input/Output

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

Formatted Input/Output with Structures

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

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

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

Unformatted Input/Output with Structures

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

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

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

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

Strings

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

String Length Issues

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

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

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

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

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

Advanced Structure Usage

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

Variable_Name.(*Tag_Index*)... ... ...

The *Tag_Index* ranges from zero to the number of fields minus one.

Note

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

Number of Structure Tags

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

Names of Structure Tags

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

Example

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

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

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

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

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

Note

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

Automatic Structure Definition

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

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

Note

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

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

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

IDL prints:

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

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

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

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

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

IDL prints:

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

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

Relaxed Structure Assignment

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

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

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

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

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

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

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

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

Using STRUCT_ASSIGN, we can make the assignment that failed using the = operator:

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

IDL prints:

```
% STRUCT_ASSIGN: SRC tag A is longer than destination.
```

```
                    The end will be clipped.
% STRUCT_ASSIGN: Destination lacks SRC tag B. Not copied.
```

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

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

IDL prints:

```
** Structure DEST, 2 tags, length=6:
   A               INT       Array[2]
   C               INT             0
```

Using Relaxed Structure Assignment

Why would you want to use Relaxed Structure Assignment? One case where this type of structure definition is very useful is in restoring object structures into an environment where the structure definition may have changed since the restored objects were saved.

Suppose you have created an application that saves data in structures. Your application may use the IDL SAVE routine to save the data structures to disk files. If you later change your application such that the definition of the data structures changes, you would not be able to restore your saved data into your application's framework without relaxed structure assignment. The RELAXED_STRUCTURE_ASSIGNMENT keyword to the RESTORE procedure allows you to make relaxed assignments in such cases.

To see how this works, try the following exercise:

1. Start IDL, create a named structure, and use the SAVE procedure to save it to a file:

```
mystruct = { STR, A:10, B:20L, C:'a string' }
SAVE, mystruct, FILE='test.dat'
```

2. Exit and restart IDL.
3. Create a new structure definition with the same name you used previously:

```
newstruct = { STR, A:20L, B:10.0, C:'a string', D:ptr_new() }
```

4. Attempt to restore the variable mystruct from the test.dat file:

```
RESTORE, 'test.dat'
```

   IDL prints:

```
% Wrong number of tags defined for structure: STR.
% RESTORE: Structure not restored due to conflict with
```

```
existing definition: STR.
```

5. Now use relaxed structure definition when restoring:

```
RESTORE, 'test.dat', /RELAXED_STRUCTURE_ASSIGNMENT
```

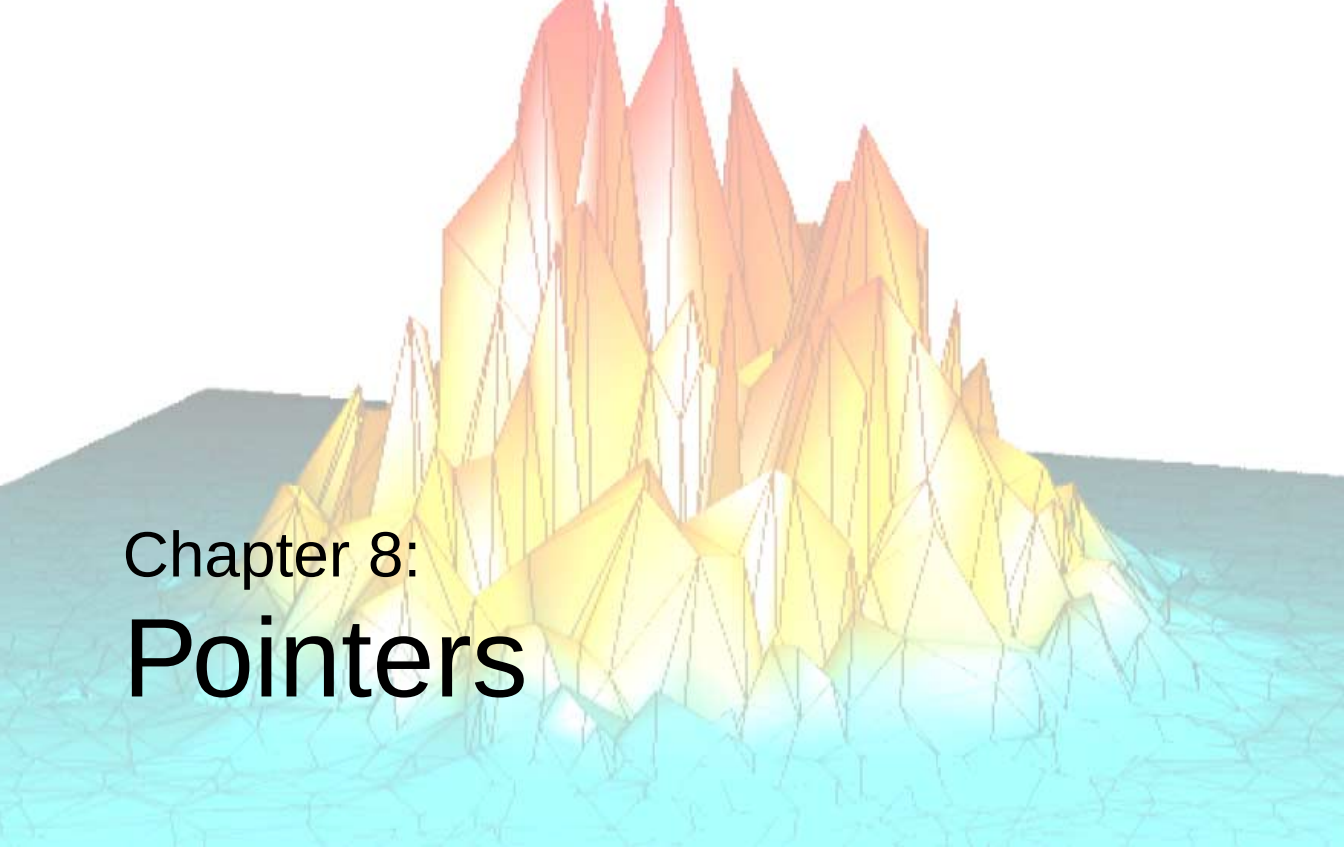
6. Check the contents of mystruct:

```
HELP, mystruct, /STRUCTURE
```

IDL prints:

```
** Structure STR, 4 tags, length=20:
   A               LONG                10
   B               FLOAT           20.0000
   C               STRING    'a string'
   D               POINTER   <NullPointer>
```

The structure in the variable mystruct now uses the definition from the new version of the STR structure, but contains the data from the old (restored) structure. In cases where the data type of a field has changed, the data type of the old data element has been converted to the new data type. Fields in the new structure definition that do not correspond to fields in the old definition contain “zero” values (zeroes for numeric fields, empty strings for string fields, null pointer or references for pointer or reference fields).



Chapter 8: Pointers

The following topics are covered in this chapter:

Overview

In order to build linked lists, trees, and other dynamic data structures, it must be possible to access variables via lightweight references that may have more than one name. Further, these names might have different lifetimes, so the lifetime of the variable that actually holds the data must be separate from the lifetime of the tokens that are used to access it.

Beginning with IDL version 5, IDL includes a new *pointer* data type to facilitate the construction of dynamic data structures. Although there are similarities between IDL pointers and machine pointers as implemented in languages such as C, it is important to understand that they are not the same thing. IDL pointers are a high level IDL language concept and do not have a direct one-to-one mapping to physical hardware. Rather than pointing at locations in computer memory, IDL pointers point at *heap variables*, which are special dynamically allocated IDL variables. Heap variables are global in scope, and exist until explicitly destroyed.

Running the Example Code

The example code used in this chapter is part of the IDL distribution. All of the files mentioned are located in the examples/doc 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" in the *IDL Reference Guide* manual 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 22, “Object Basics” for more information on IDL objects.) In IDL documentation of pointers and objects, heap variables accessible via pointers are called *pointer heap variables*, and heap variables accessible via object references are called *object heap variables*.

Note

Pointers and object references have many similarities, the strongest of which is that both point at heap variables. It is important to understand that they are not the same type, and cannot be used interchangeably. Pointers and object references are used to solve different sorts of problems. Pointers are useful for building dynamic data structures, and for passing large data around using a lightweight token (the pointer itself) instead of copying data. Objects are used to apply object oriented design techniques and organization to a system. It is, of course, often useful to use both in a given program.

Heap variables are global in scope, but do not suffer from the limitations of COMMON blocks. That is, heap variables are available to all program units at all times. (Remember, however, that IDL variables containing pointers to heap variables are *not* global in scope and must be declared in a COMMON block if you want to share them between program units.)

Heap variables:

- Facilitate object oriented programming.
- Provide full support for Save and Restore. Saving a pointer or object reference automatically causes the associated heap variable to be saved as well. This means that if the heap variable contains a pointer or object reference, the heap variables they point to are also saved. Complicated self-referential data structures can be saved and restored easily.
- Are manipulated primarily via pointers or object references using built in language operators rather than special functions and procedures.
- Can be used to construct arbitrary, fully general data structures in conjunction with pointers.

Note

If you have used versions of IDL prior to version 5, you may be familiar with *handles*. Because IDL pointers provide a more complete and robust way of building

dynamic data structures, RSI recommends that you use pointers rather than handles when developing new code. See Appendix I, “Obsolete Features” in the *IDL Reference Guide* manual for a discussion of RSI’s 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 22, “Object Basics” for a discussion of object creation.) Copying a pointer or object reference *does not* create a new heap variable. This is markedly different from the way IDL handles “regular” variables. For example, with the statement:

```
A = 1.0
```

you create a new IDL floating-point variable with a value of 1.0. The following statement:

```
B = A
```

creates a second variable with the same type and value as A.

In contrast, if you create a new heap variable with the following command:

```
C = PTR_NEW(2.0d)
```

the variable C contains not the double-precision floating-point value 2.0, but a pointer to a heap variable that contains that value. Copying the variable C with the following statement:

```
D = C
```

does not create another heap variable, but rather creates a second pointer to the same heap variable. In this example, the HELP command would reveal:

```
% At  $MAIN$
A               FLOAT     =       1.00000
B               FLOAT     =       1.00000
C               POINTER   = <PtrHeapVar1>
D               POINTER   = <PtrHeapVar1>
```

The variables C and D are both pointers to the same heap variable. (The actual name assigned to a heap variable is arbitrary.) Changing the value stored in the heap variable would be reflected when dereferencing either C or D (dereferencing is discussed in “Dereference” on page 175).

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 180 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 179.

Saving and Restoring Heap Variables

The SAVE and RESTORE procedures work for heap variables just as they work for all other supported types. When IDL saves a pointer or object reference in a save file, it recursively saves the heap variables that are referenced by that pointer or object reference. SAVE handles circular data structures correctly. You can build a large, complicated, self-referential data structure, and then save the entire construct with a call to SAVE to save the single pointer or object reference that points to the head of the structure. For example, you can save a pointer to the root of a binary tree and the entire tree will be saved.

The internal identifier of a given heap variable is dynamically allocated at run time, and will differ between IDL sessions. As a result, the RESTORE operation maps all saved pointers and object references to their new values in the current session.

Pointer Heap Variables

Pointer heap variables are IDL heap variables that are accessible only via *pointers*. While there are many similarities between object references and pointers, it is important to understand that they are not the same type, and cannot be used interchangeably. Pointer heap variables are created using the PTR_NEW and PTRARR functions. For more information on objects, see Chapter 22, “Object Basics”.

IDL Pointers

As illustrated above, you must use a special IDL routine to create a pointer to a heap variable. Two routines are available: PTR_NEW and PTRARR. Before discussing these functions, however, it is useful to examine the concept of a null pointer.

Null Pointers

The *Null Pointer* is a special pointer value that is guaranteed to never point at a valid heap variable. It is used by IDL to initialize pointer variables when no other initializing value is present. It is also a convenient value to use at the end nodes in data structures such as trees and linked lists.

It is important to understand the difference between a null pointer and a pointer to an undefined or invalid heap variable. The second case is a valid pointer to a heap variable that does not currently contain a usable value. To make the difference clear, consider the following IDL statements:

```
;The variable A contains a null pointer.
A = PTR_NEW()
;The variable B contains a pointer to a heap variable with an
;undefined value.
B = PTR_NEW(/ALLOCATE_HEAP)

HELP, A, B, *B
```

IDL prints:

```
A               POINTER  = <NullPointer>
B               POINTER  = <PtrHeapVar1>
<PtrHeapVar1>   UNDEFINED = <Undefined>
```

The primary difference is that it is possible to write a useful value into a pointer to an undefined variable, but this is never possible with a null pointer. For example, attempt to assign the value 34 to the null pointer:

```
*A = 34
```

IDL prints:

```
% Unable to dereference NULL pointer: A.
% Execution halted at:  $MAIN$
```

Assign the value 34 to a previously-undefined heap variable:

```
*B = 34
PRINT, *B
```

IDL prints:

```
34
```

Similarly, the null pointer is not the same thing as the result of PTR_NEW(0). PTR_NEW(0) returns a pointer to a heap variable that has been initialized with the integer value 0.

The PTR_NEW Function

Use the PTR_NEW function to create a single pointer to a new heap variable. If you supply an argument, the newly-created heap variable is set to the value of the argument. For example, the command:

```
ptr1 = PTR_NEW(FINDGEN(10))
```

creates a new heap variable that contains the ten-element floating point array created by FINDGEN, and places a pointer to this heap variable in ptr1.

Note that the argument to PTR_NEW can be of any IDL data type, and can include any IDL expression, including calls to PTR_NEW itself. For example, the command:

```
ptr2 = PTR_NEW({name:'', next:PTR_NEW()})
```

creates a pointer to a heap variable that contains an anonymous structure with two fields: the first field is a string, the second is a pointer. We will develop this idea further in the examples at the end of this chapter.

If you do not supply an argument, the newly-created pointer will be a null pointer. If you wish to create a new heap variable but do not wish to initialize it, use the ALLOCATE_HEAP keyword.

See “PTR_NEW” in the *IDL Reference Guide* manual for further details.

The PTRARR Function

Use the PTRARR function to create an array of pointers of up to eight dimensions. By default, every element of the array created by PTRARR is set to the null pointer. For example:

```
;Create a 2 by 2 array of null pointers.
ptarray = PTRARR(2,2)

;Display the contents of the ptarray variable, and of the first
;array element.
HELP, ptarray, ptarray(0,0)
```

IDL prints:

```
PTARR           POINTER   = Array(2, 2)
<Expression>    POINTER   = <NullPointer>
```

If you want each element of the array to point to a new heap variable (as opposed to being a null pointer), use the ALLOCATE_HEAP keyword. Note that in either case, you will need to initialize the array with another IDL statement.

See "PTRARR" in the *IDL Reference Guide* manual 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" in the *IDL Reference Guide* manual), or do manual "Garbage Collection" and use the HEAP_GC command to destroy all inaccessible heap variables.

Warning

Object reference heap variables are subject to the same problems as pointer heap variables. See "OBJ_VALID" in the *IDL Reference Guide* manual for more information.

The HEAP_GC procedure causes IDL to hunt for all unreferenced heap variables and destroy them. It is important to understand that this is a potentially computationally expensive operation, and should not be relied on by programmers as a way to avoid writing careful code. Rather, the intent is to provide programmers with a debugging aid when attempting to track down heap variable leakage. In conjunction with the VERBOSE keyword, HEAP_GC makes it possible to determine when variables have leaked, and it provides some hint as to their origin.

Warning

HEAP_GC uses a recursive algorithm to search for unreferenced heap variables. If HEAP_GC is used to manage certain data structures, such as large linked lists, a potentially large number of operations may be pushed onto the system stack. If so many operations are pushed that the stack runs out of room, IDL will crash.

General reference counting, the usual solution to such leaking, is too slow to be provided automatically by IDL, and careful programming can easily avoid this pitfall. Furthermore, implementing a reference counted data structure on top of IDL pointers is easy to do in those cases where it is useful, and such reference counting could take advantage of its domain specific knowledge to do the job much faster than the general case.

Another approach would be to write allocation and freeing routines—layered on top of the PTR_NEW and PTR_FREE routines—that keep track of all outstanding pointer allocations. Such routines might make use of pointers themselves to keep track of the allocated pointers. Such a facility could offer the ability to allocate pointers in named groups, and might provide a routine that frees all heap variables in a given group. Such an operation would be very efficient, and is easier than reference counting.

Pointer Validity

Use the PTR_VALID function to verify that one or more pointer variables point to valid and currently existing heap variables, or to create an array of pointers to existing heap variables. If supplied with a single pointer as its argument, PTR_VALID returns TRUE (1) if the pointer argument points at a valid heap variable, or FALSE (0) otherwise. If supplied with an array of pointers, PTR_VALID returns an array of TRUE and FALSE values corresponding to the input array. If no argument is specified, PTR_VALID returns an array of pointers to all existing pointer heap variables. For example:

```
;Create a new pointer and heap variable.
A = PTR_NEW(10)

IF PTR_VALID(A) THEN PRINT, "A points to a valid heap variable." $
    ELSE PRINT, "A does not point to a valid heap variable."
```

IDL prints:

```
A points to a valid heap variable.
```

For example:

```
;Destroy the heap variable.
PTR_FREE, A

IF PTR_VALID(A) THEN PRINT, "A points to a valid heap variable." $
    ELSE PRINT, "A does not point to a valid heap variable."
```

IDL prints:

```
A does not point to a valid heap variable.
```

See “PTR_VALID” in the *IDL Reference Guide* manual 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 179.

See "PTR_FREE" in the *IDL Reference Guide* manual 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" in the *IDL Reference Guide* manual for further details.

Pointer Examples

Pointers are useful in building dynamic memory structures, such as linked lists and trees. The following examples demonstrate how pointers are used to build several types of dynamic structures. Note that the purpose of these examples is to illustrate simply and clearly how pointers are used. As such, they may not represent the “best” or most efficient way to accomplish a given task. Readers interested in learning more about efficient use of data structures are urged to consult any good text on data structures.

Creating a Linked List

The following example uses pointers to create and manipulate a linked list. One procedure reads string input from the keyboard and creates a list of pointers to heap variables that have the strings as their values. Another procedure prints the strings, given the pointer to the beginning of the linked list. A third procedure uses a modified “bubble sort” algorithm to reorder the values so the strings are in alphabetical order.

Creating the List

The following program prompts the user to enter a series of strings from the keyboard. After reading each string, it creates a new heap variable containing a list element—an anonymous structure with two fields; one to hold the string data and one to hold a pointer to the next list element. Any number of strings can be entered. When the user is finished entering strings, the program can be exited by entering a period by itself at the “Enter string:” prompt.

The text of the program shown below can be found in the file ptr_read.pro in the examples/doc subdirectory of the IDL distribution.

```
;PTR_READ accepts one argument, a named variable in which to return
;the pointer that points at the beginning of the list.
PRO ptr_read, first

;Initialize the input string variable.
newstring = ''

;Create an anonymous structure to contain list elements. Note that
;the next field is initialized to be a null pointer.
llist = {name:'', next:PTR_NEW()}

;Print instructions for this program.
PRINT, 'Enter a list of names.'
PRINT, 'Enter a period (.) to stop list entry.'
```

```
;Continue accepting input until a period is entered.
WHILE newstring NE "." DO BEGIN

   READ, newstring, PROMPT='Enter string: '
;Read a new string from the keyboard.

;Check to see if a pointer called first exists. If not, this is
;the first element. Create a pointer called first and initialize
;it to be a list element. Create a second pointer to the heap
;variable pointed at by first.
   IF newstring NE '.' THEN BEGIN
     IF ~(PTR_VALID(first)) THEN BEGIN
       first = PTR_NEW(llist)
       current = first
     ENDIF

     ;Create a pointer to the next list element.
     next = PTR_NEW(llist)

     ;Set the name field of current to the input string.
     (*current).name = newstring

     ;Set the next field of current to the pointer to the next list
     ;element.
     (*current).next = next

     ;Store the "current" pointer as the "last" pointer.
     last = current

     ;Make the "next" pointer the "current" pointer.
     current = next

   ENDIF
ENDWHILE

;Set the next field of the last element to the null pointer.
IF PTR_VALID(last) THEN (*last).next = PTR_NEW()

;End of PTR_READ program.
END
```

Run the PTR_READ program by entering the following command at the IDL prompt:

```
ptr_read, first
```

Type a string, press Return, and the program prompts for another string. You can enter as many strings as you want. Each time a string is entered, PTR_READ creates

a new list element with that string as its value. For example, you could enter the following three strings (used in the rest of this example):

```
Enter a list of names.
Enter a period (.) to stop list entry.
Enter string: wilma
Enter string: biff
Enter string: cosmo
Enter string: .
```

The following figure shows one way of visualizing the linked list that we've created.

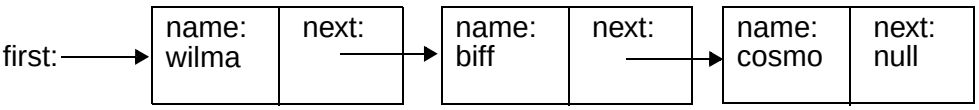


Table 8-1: One way of visualizing the linked list created by the PTR_READ procedure

Printing the Linked List

The next program in our example accepts the pointer to the first element of the linked list and prints all the values in the list in order. To illustrate how the list is linked, we will also print the name of the heap variable that contains each element, and the name of the heap variable in the next field of that element.

The text of the program shown below can be found in the file ptr_print.pro in the examples/doc subdirectory of the IDL distribution.

```
;PTR_PRINT accepts one argument, a pointer to the first element of
;a linked list returned by PTR_READ. Note that the PTR_PRINT
;program does not need to know how many elements are in the list,
;nor does it need to explicitly know of any pointer other than the
;first.
PRO ptr_print, first

;Create a second pointer to the heap variable pointed at by first.
current = first

;PTR_VALID returns 0 if its argument is not a valid pointer. Note
;that the null pointer is not a valid pointer.
WHILE PTR_VALID(current) DO BEGIN

   ;Print the list element information.
   PRINT, current, ', named ', (*current).name, $
     ', has a pointer to: ', (*current).next
```

```
   ;Set current equal to the pointer in its own next field.
   current = (*current).next

ENDWHILE

;End of PTR_PRINT program.
END
```

If we run the PTR_PRINT program with the list generated in the previous example:

```
IDL> ptr_print, first
```

IDL prints:

```
<PtrHeapVar1>, named wilma, has a pointer to: <PtrHeapVar2>
<PtrHeapVar2>, named biff, has a pointer to: <PtrHeapVar3>
<PtrHeapVar3>, named cosmo, has a pointer to: <NullPointer>
```

A Simple Sorting Routine for the Linked List

The next example program takes a list generated by PTR_READ and moves the values so that they are in alphabetical order. The sorting algorithm used in this program is a variation on the classic “bubble sort”. However, instead of starting with the last element in the list and letting lower values “rise” to the top, this example starts at the top of the list and lets higher (“heavier”) values “sink” to the bottom of the list. Note that this is not a very efficient sorting algorithm and is shown as an illustration because of its simplicity. For real sorting applications, use IDL’s SORT function.

The text of the program shown below can be found in the file ptr_sort.pro in the examples/doc subdirectory of the IDL distribution.

```
;PTR_SORT accepts one argument, a pointer to the first element of a
;linked list returned by PTR_READ. Note that the PTR_SORT program
;does not need to know how many elements are in the list, nor does
;it need to explicitly know of any pointer other than the first.
pro ptr_sort, first

;Initialize swap flag.
swap = 1

;Create an anonymous structure to contain list elements. Note that
;the next field is initialized to be a pointer.
llist = {name:'', next:PTR_NEW()}

;Create a pointer to this structure, to be used as "swap space."
junk = ptr_new(llist)
```

```
;Continue the sorting until no swaps are made. If no adjacent
;elements need to be swapped, the list is in alphabetical order.
WHILE swap NE 0 DO BEGIN

   ;Create a second pointer to the heap variable pointed at by
   ;first.
   current = first

   ;Create another pointer to the heap variable held in the next
   ;field of current.
   next = (*current).next

   ;Set swap flag.
   swap = 0

   ;Continue the sorting until next is no longer a valid pointer.
   ;Note that the null pointer is not a valid pointer.
   WHILE PTR_VALID(next) DO BEGIN

     ;Get values to compare.
     value1 = (*current).name
     value2 = (*next).name

     ;Compare values and exchange if first is greater than second.
     IF (value1 GT value2) THEN BEGIN

       ;Use the "swap space" pointer to exchange the name fields of
       ;current and next.
       (*junk).name = (*current).name
       (*current).name = (*next).name
       (*next).name = (*junk).name

       ;Set current to next to advance through the list.
       current = next

       ;Reset swap flag.
       swap = 1

     ;If value1 is less than value2, set current to next to advance
     ;through the list.
     ENDIF ELSE current = next

     ;Redefine next pointer.
     next = (*current).next
   ENDWHILE
ENDWHILE
END
```

To run the PTR_SORT routine with the list from our previous examples as input, enter:

```
ptr_sort, first
```

We can see the results of the sorting by calling the PTR_PRINT routine again:

```
ptr_print, first
```

IDL prints:

```
<PtrHeapVar1>, named biff, has a pointer to: <PtrHeapVar2>
<PtrHeapVar2>, named cosmo, has a pointer to: <PtrHeapVar3>
<PtrHeapVar3>, named wilma, has a pointer to: <NullPointer>
```

and we see that now the names are in alphabetical order.

Example Files—Using Pointers to Create Binary Trees

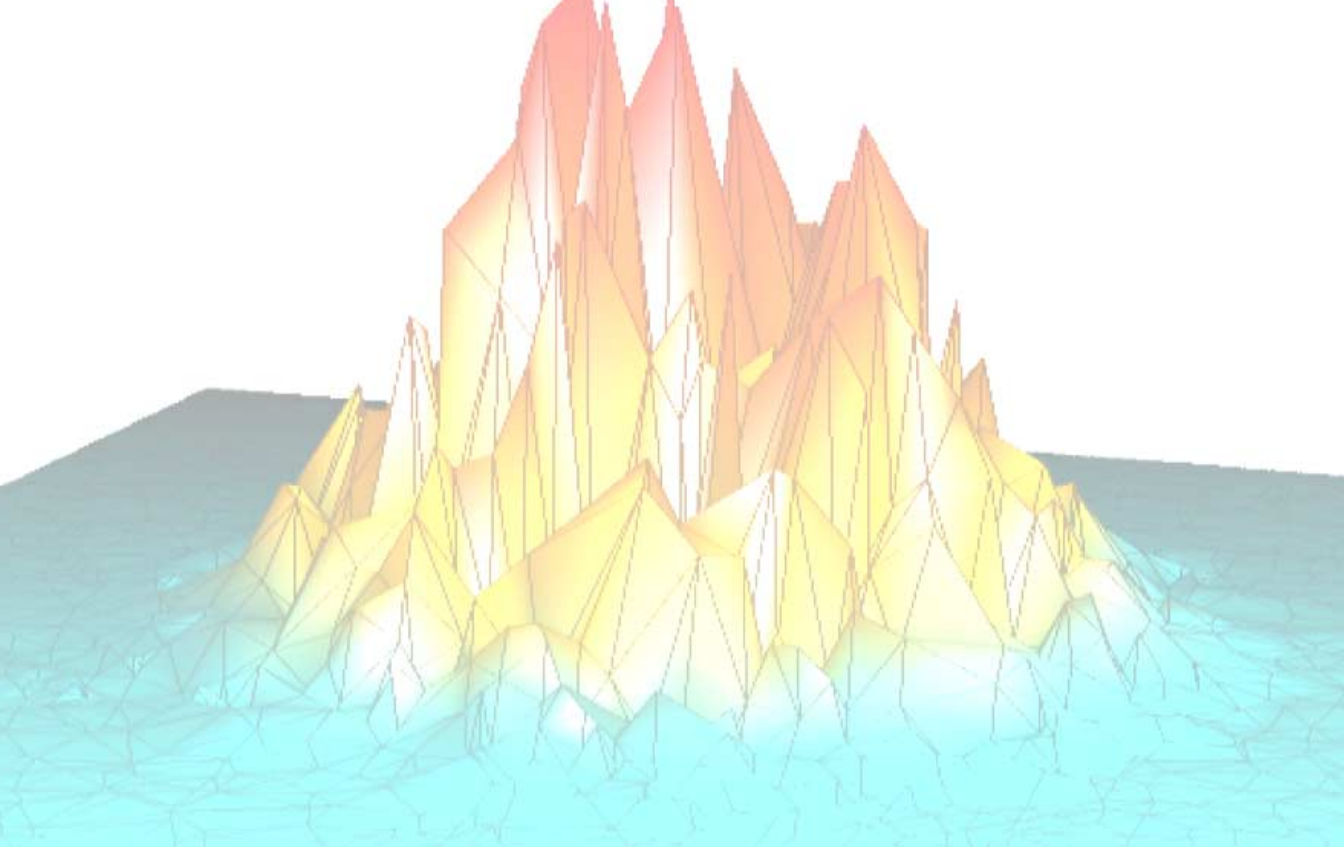
Two more-complicated example programs demonstrate the use of IDL pointers to create and search a simple tree structure. These files, named `idl_tree.pro` and `tree_example.pro`, can be found in the examples/doc subdirectory of the IDL distribution.

To run the tree examples, enter the following commands at the IDL prompt:

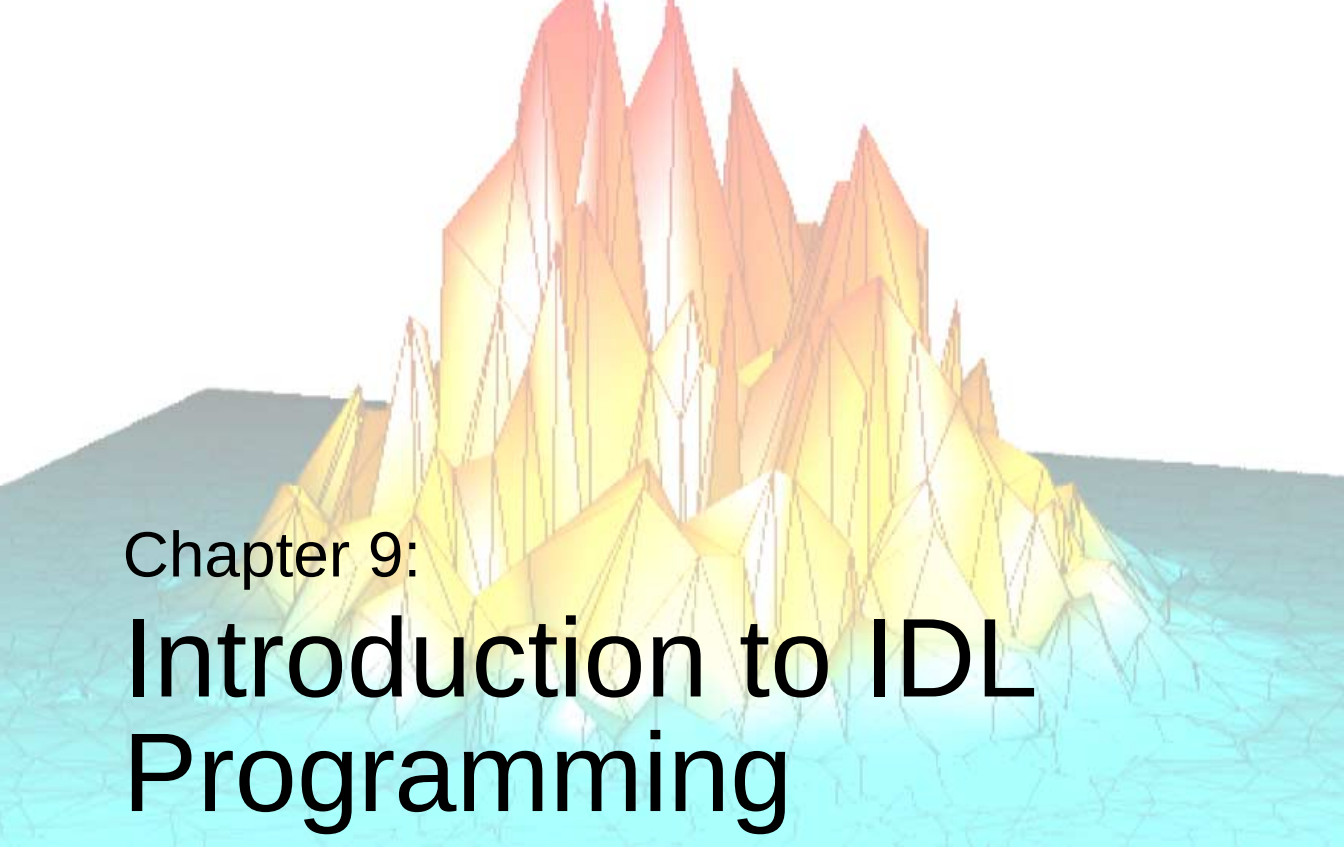
```
;Compile the routines in idl_tree. The example routine calls the
;routines defined in this file.
.run idl_tree

;Run the tree_example.
tree_example
```

The TREE_EXAMPLE and IDL_TREE routines create a binary tree with ten nodes whose values are structures that contain random values for two fields, “Time” and “Data”. The TREE_EXAMPLE routine then prints the tree sorted by both time and data. It then searches for and deletes the nodes containing the fourth and second data values. The resulting 8-node trees are again printed in both time and data order.

A detailed explication of the TREE_EXAMPLE and IDL_TREE routines is beyond the scope of this chapter. Interested users should examine the files, starting with tree_example.pro, to see how the trees are created and searched.



Part II: Basics of IDL Programming



Chapter 9: Introduction to IDL Programming

The following topics are covered in this chapter:

What is an IDL Program?

There are three types of IDL programs: main-level programs, include files, and program files.

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 Chapter 3, "Using the IDL Editor" in the *Using IDL* manual for details on using the IDL Editor.

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 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 in 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.

To create and run a simple main-level program, do the following:

1. Start IDL
2. At the IDL command line, enter the following:

   ```
   A = 2
   ```

3. Enter `.RUN` at the IDL command line. The command line prompt changes from `IDL>` to `-`.
4. Enter the following:

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

5. This creates a main-level program, which compiles and executes. IDL prints 4.
6. Enter `.GO` at the IDL command line. The main-level program is executed again, and now IDL prints 8.

Include Files

An include file contains one or more IDL statements or commands. Each line of the include file is read and executed before proceeding to the next line. This makes include files different form main-level programs, in which the main-level program is compiled as a unit before being executed, and program files, in which all modules contained in the file are compiled as an unit before being executed. A file created by the JOURNAL routine is an example of an include file. For information on running include files, see Chapter 10, "Executing Batch Jobs in IDL" in the *Using IDL* manual.

Program Files

Most IDL applications are in the form of program files. Program files are text files that contain IDL procedures and/or functions:

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

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`. 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` extension.

Procedures and functions can also contain *arguments* and *keywords*. Arguments allow variables to be inputted into and/or outputted from a procedure or function. Keywords are usually used to set specific parameters pertaining to a procedure or function.

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)) THEN PRINT, 'Hello World From ' + $
      name 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, supplied via the *name* argument.

Procedures and functions can also be referred to as *routines*. An IDL program file may contain one or many routines, which can be a mix of procedures and functions. These routines can be written into an IDL program file using the IDL Editor.

Creating a Simple Program

In this section, we'll create a simple "Hello World" program consisting of two `.pro` files:

1. Start the IDLDE.
2. Start the IDL Editor by selecting **File** → **New** or clicking the **New File** button on the toolbar.
3. Type the following in the IDL Editor window:

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

4. To save the file, select **File** → **Save** or click **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).
5. Open a new Editor window by selecting **File** → **New**, and enter the following code:

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

6. Save the file as `hello_who.pro` in the main IDL directory.

We now have a simple program consisting of a user-defined procedure, which calls a user-defined function.

Compiling and Running 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 PLOT) 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 function or procedure and then compile it. When you enter the name of an uncompiled user-defined procedure at the command line or call the procedure from another procedure, IDL searches the current directory for *filename*.pro, then *filename*.sav, where *filename* is the name of the procedure. If no file is found in the current directory, IDL searches each directory specified by !PATH. (For more on the IDL path, see "!PATH" in the *IDL Reference Guide* manual.) If a file is found, IDL automatically compiles the contents and executes the function or procedure that has the same name as the file specified (excluding the suffix).

There are several ways to compile a procedure or function:

- If the file is open in the IDL Editor, select **Compile** from the **Run** menu or click the **Compile** button on the toolbar.
- Use the `.COMPILE` executive command at the IDL command line.
- Enter the name of the procedure or function at the IDL command line. Multiple procedures and/or functions can be defined in the same `.pro` file, so if the file defines more than one procedure or function, only the procedure or function with the name entered at the command line will be compiled (and subsequently executed). 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.

If you select the **Compile** button on the IDLDE toolbar or you enter `.COMPILE proc1` at the command line, all three procedures will be compiled. You can then enter either `proc1`, `proc2`, or `proc3` at the command line to execute the corresponding procedure.

In our "Hello World" example, we have 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.

In general, the name of the IDL program file should be the same as the name of the last procedure or function within this file. This last routine is usually the main routine, which calls all the other routines within the IDL program file. 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.

Many 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 Output Log 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.
```

Since these routines are now compiled, you can run XLOADCT:

```
IDL> XLOADCT
% Compiled module: XREGISTERED.
% Compiled module: LOADCT.
% Compiled module: FILEPATH.
% Compiled module: CW_BGROUP.
% Compiled module: XMANAGER.
```

The remaining compiled modules are other IDL program files contained within the distribution. These files (routines) are called within the XLOADCT routine.

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 by selecting

its name from the Functions/Procedures Menu. See "Functions/Procedures Menu" in Chapter 3 of the *Using IDL* manual for more information.

Compilation Errors

If an error occurs during compilation, the error is reported in the Output Log of the IDLDE. 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 the following error in the IDL Output Log:

```
IDL> .COMPILE procedure_without_END

% End of file encountered before end of program.
% 1 Compilation errors in module PROCEDURE_WITHOUT_END.
```

Note

Under Microsoft Windows, the IDL Editor window displays a red dot to the left of each line that contains an error.

Commenting Your IDL Code

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 annotate programs with comments. There are no execution-time or space penalties for comments in IDL.

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.
```

Saving Compiled IDL Programs

The SAVE procedure can be used to quickly save simple IDL routines in a binary format that can be shared with other IDL users. This section covers how to create a `.sav` file of simple routine containing a call to a secondary `.sav` file containing image variable data. Using SAVE works well with simple routines.

Note

Variables and routines cannot be stored in the same `.sav` file.

If your program or utility requires multiple program (`.pro`) files, each procedure or function used by your program must resolved and contained in a `.sav` file. When multiple routines are required by your IDL application, you have the following options:

- Include all routines in a main `.sav` file that is restored first. This makes all routines available without having to restore any additional `.sav` files. The easiest way to do this is to add all `.pro` files to an IDL Project and build the project, which creates a single `.sav` file. See Chapter 20, “Creating IDL Projects”.
- Create a separate `.sav` file for each routine used by your application. Assuming each `.sav` file has the same name as the procedure or function it contains, this allows you to call each routine without having to explicitly restore its `.sav` file. This is because IDL will search for the `.sav` file and restore it automatically when it encounters the first call to the routine.

If your program also contains additional variable data, you must create a separate `.sav` file containing the variable data. Variable data must be restored before any routine attempts to use the variables contained in the file. If you are working with multiple data, image and program files, consider using the IDL Project interface to create a single `.sav` file. See Chapter 20, “Creating IDL Projects” for more information. Alternately, you will need to restore the variables using the RESTORE procedure before referencing the variables in any IDL program. See “Restoring Compiled IDL Programs and Data” on page 209 for more information.

Note

A `.sav` file containing data will always be restorable. However, `.sav` files that contain IDL procedures, functions, and programs are not always portable between different versions of IDL. In this case, you will need to recompile your original `.pro` files and re-create `.sav` files using the current version of IDL.

Creating a .sav File of a Simple Routine

The following example creates two `.sav` files. One `.sav` file will contain variable data, an image file. This `.sav` file is then restored in the main `.sav` file which uses a simple call to the ARROW procedure to point out an area of interest within the image. To create these files, complete the following steps:

1. Start a fresh session of IDL to avoid saving unwanted session information.
2. Open the image file of a 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. Use the SAVE procedure to save the *image* variable within a `.sav` file by entering the following:

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

   This stores the `.sav` 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.

4. Create the following IDL program that first restores the *image* variable contained within the secondary `.sav` file, `imagefile.sav`. 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

END
```

5. Save the file as `draw_arrow.pro`. Next, create a `.sav` file of this program file.
6. Exit and restart IDL or enter .FULL_RESET_SESSION at the IDL prompt before creating a `.sav` file to avoid saving unwanted session information.
7. Open `draw_arrow.pro` and compile it by entering:

```
.COMPILE draw_arrow
```

8. Use RESOLVE_ALL 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.

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

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

Note

When the name of the `.sav` file is the same as the main level program, it can be called from another routine or restored from the IDL command line by simply stating the name of the file, minus the `.sav` extension. This method of naming `.sav` files is recommended.

Customizing and Saving an ASCII Template

When importing an ASCII data file into IDL, you must first describe the format of the data using the interactive ASCII_TEMPLATE function. If you have a number of ASCII files that have the same format, you can create and save a customized ASCII

template using the SAVE procedure. After creating a `.sav` file of your custom template, you can avoid having to repeatedly define the same fields and records when reading in ASCII files that have the same structure.

1. At the IDL command line, enter the following to create the variable *plotTemplate*, which will contain your custom ASCII template:

   ```
   plotTemplate = ASCII_TEMPLATE( )
   ```

   A dialog box appears, prompting you to select a file.

2. Select `plot.txt` located in the `examples/data` directory.

   Note

   Another way to import ASCII data is to use the **Import ASCII File** toolbar button on the IDLDE toolbar. To use this feature, simply click the button and select `plot.txt` from the file selection dialog.

3. After selecting the file, the **Define Data Type/Range** dialog appears. First, choose the field type. Since the data file is delimited by tabs (or whitespace) select the **Delimited** button. In the **Data Starts at Line** field, specify to begin reading the data at line 3, not line 1, since there are two comment lines at the beginning of the file. Click **Next** to continue.

4. In the **Define Delimiter/Fields** dialog box, select **Tab** as the delimiter between data elements since it is known that tabs were used in the original file. Click **Next**.

5. In the **Field Specification** dialog box, name each field as follows:

   - Click on the first row (row 1). In the **Name** field, enter `time`.
   - Select the second row and enter `temperature1`.
   - Select the third row and enter `temperature2`.

6. Click **Finish**.

7. Type the following line at the IDL command line to read in the `plot.txt` file using the custom template, `plotTemplate`:

   ```
   PLOT_ASCII = READ_ASCII(FILEPATH('plot.txt', SUBDIRECTORY = $
      ['examples', 'data']), TEMPLATE = plotTemplate)
   ```

8. Enter the following line to print the `plot.txt` file data:

   ```
   PRINT, PLOT_ASCII
   ```

The file contents are printed in the Output Log window. Your output will resemble the following display.

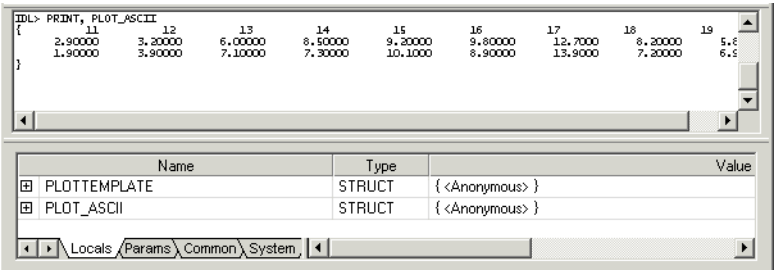


Figure 9-1: PLOT_ASCII Printout

9. Create a binary `.sav` file of your custom template by entering the following:

```
SAVE, plotTemplate, FILENAME='myPlotTemplate.sav'
```

10. To restore the template so that you can read another ASCII file, enter:

```
RESTORE, 'myPlotTemplate.sav'
```

This file contains your custom ASCII template information stored in the structure variable, `plotTemplate`.

Note

If you are attempting to restore a file that is not in your current working directory or the IDL search path, you will need to specify a path to the file. See "RESTORE" in the *IDL Reference Guide* manual for more information.

11. After restoring your custom template, you can read another ASCII file that is delimited in the same way as the original file by using the READ_ASCII function and specifying `plotTemplate` for the TEMPLATE:

```
PLOT_ASCII = READ_ASCII(FILEPATH('plot.txt', $
   SUBDIRECTORY = ['examples', 'data']), $
   TEMPLATE = plotTemplate)
```

12. Enter the following to display the contents of the file using the customized ASCII template structure previously defined using the dialog.

```
PRINT, PLOT_ASCII
```

Saving and Restoring the XROI Utility and Image ROI Data

You can easily share your own IDL routines or utilities with other IDL users by using the SAVE routine to create a binary file of your compiled code. The following example creates a `.sav` file of the XROI utility (a `.pro` file) and from within this file, restores a secondary `.sav` file containing selected regions of interest.

1. Type `XROI` at the command line to open the XROI utility.
2. In the file selection dialog, select `mineral.png` located in the `examples/data` directory.
3. Select the **Draw Polygon** toolbar button and roughly outline the three large, angular areas of the image.
4. Select **File** → **Save ROIs** and name the file `mineralROI.sav`. This creates a `.sav` file containing the regions of interest selected within the image.
5. In an IDL Editor or text editor, enter the following routine:

```
PRO myXRoi

; Restore ROI object data by specifying a value for the
; RESTORED_OBJECTS keyword.
RESTORE, 'mineralROI.sav', RESTORED_OBJECTS = myROI

; Open XROI, specifying the previously defined value for the
; restored object data as the value for "REGIONS_IN".
XROI, READ_PNG(FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data'])), $
   REGIONS_IN = myROI, /BLOCK

END
```

   Save the routine as `myXRoi.pro`
6. Exit and restart IDL or enter .FULL_RESET_SESSION at the IDL command line before creating a `.sav` file to avoid saving unwanted session information.
7. After re-opening the `myXRoi` routine, compile the program you just created:

```
.COMPILE myXRoi.pro
```

8. Use RESOLVE_ALL 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 class methods, nor 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.

9. Create a `.sav` file named `myXRoi.sav`, containing all of the XROI utility routines. When the SAVE procedure is called with the ROUTINES keyword and no arguments, it creates a `.sav` file containing all currently compiled routines. Because the routines associated with the XROI utility are the only ones that are currently compiled in our IDL session, we can create a `.sav` file as follows:

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

10. It is not necessary to use RESTORE to open `myXRoi.sav`. If the main level routine is named the same as the `.sav` file, and all necessary files (in this case, `mineralROI.sav` and `myXRoi.sav`) are stored in the current working directory or the IDL search path, simply type the name of the file, minus the `.sav` extension, at the command line:

```
myXRoi
```

The following figure will appear, showing the selected regions of interest.

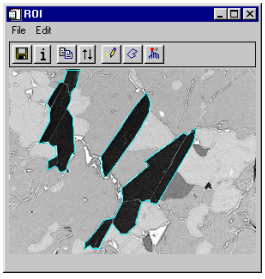


Figure 9-2: Example of Restoring the XROI Utility and ROI Image Data

Restoring Compiled IDL Programs and Data

This section covers various ways to restore files created with the SAVE procedure, which typically have a `.sav` extension. The options for restoring these files include the following:

- Restoring `.sav` files from the command line. This method is typically used to restore and run `.sav` files containing an IDL routine with the same name as the `.sav` file. See "Restoring .sav Files from the Command Line" on page 209.
- Using the RESTORE procedure to explicitly restore a `.sav` file. You must use RESTORE to restore variable data. You may also need to use RESTORE if your `.sav` file contains multiple routines or a routine that has a different name than the `.sav` file. See "Using RESTORE to Explicitly Restore a .sav File" on page 209.
- Restoring a secondary `.sav` file from within a main `.sav` file. This is commonly done to restore variable data before it is needed by the IDL program or routine. See "Creating a .sav File of a Simple Routine" on page 203 for an example.

Restoring .sav Files from the Command Line

To restore a `.sav` file containing an IDL program file with the same name as the `.sav` file, simply type the name of the file, minus the `.sav` extension, at the IDL command line. For example, to restore the file, `draw_arrow.sav`, containing the `draw_arrow` routine (created in the previous section, "Creating a .sav File of a Simple Routine" on page 203), enter the following at the command line:

```
draw _arrow
```

When a file is specified by typing only the filename at the IDL prompt, IDL searches the current directory for `draw_arrow.pro` and then for `draw_arrow.sav`. If no file is found in the current directory, IDL searches in the same way in each directory specified by !PATH. See "!PATH" in the *IDL Reference Guide* manual for more information. If the file is not located in your current working directory or IDL search path, IDL will report that the file cannot be found. See "RESTORE" in the *IDL Reference Guide* manual for information on how to define a path to the file.

Using RESTORE to Explicitly Restore a .sav File

You must use the RESTORE procedure to explicitly restore any file containing variable data. In the previous section, "Creating a .sav File of a Simple Routine" on

page 203, two .sav files were created; imagefile.sav and draw_arrow.sav. The imagefile.sav file contains image variable data. To explicitly restore the image data, enter the following at the IDL command line:

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

Information about the variable, *image*, which is contained within the .sav file, appears in the IDLDE Variable Watch window.

Note

If the file you are attempting to restore is not located in your current working directory or the IDL search path, you will need to specify a path to the file. See "RESTORE" in the *IDL Reference Guide* manual for information on how to define a path to the file.

You also need to use the RESTORE procedure when your .sav file contains multiple routines or a routine that has a different name that the .sav file. In such cases, you must use RESTORE to restore the main .sav file before calling any of the routines within the .sav file. For example, suppose you have a .sav file containing the draw_arrow routine created in the previous section, but you have named the file myarrow.sav (instead of draw_arrow.sav) as shown in the following statement:

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

In this case you must use the RESTORE procedure to restore the .sav file before calling the draw_arrow routine.

```
RESTORE, 'myarrow.sav', /VERBOSE
```

This restores all routines associated with the file. Use the VERBOSE keyword to print an informative message about each restored object. To run the draw_arrow routine, now enter the following at the command line:

```
draw_arrow
```

Note on IDL 5.4 SAVE Files

With IDL 5.4, RSI released a version of IDL that was 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.

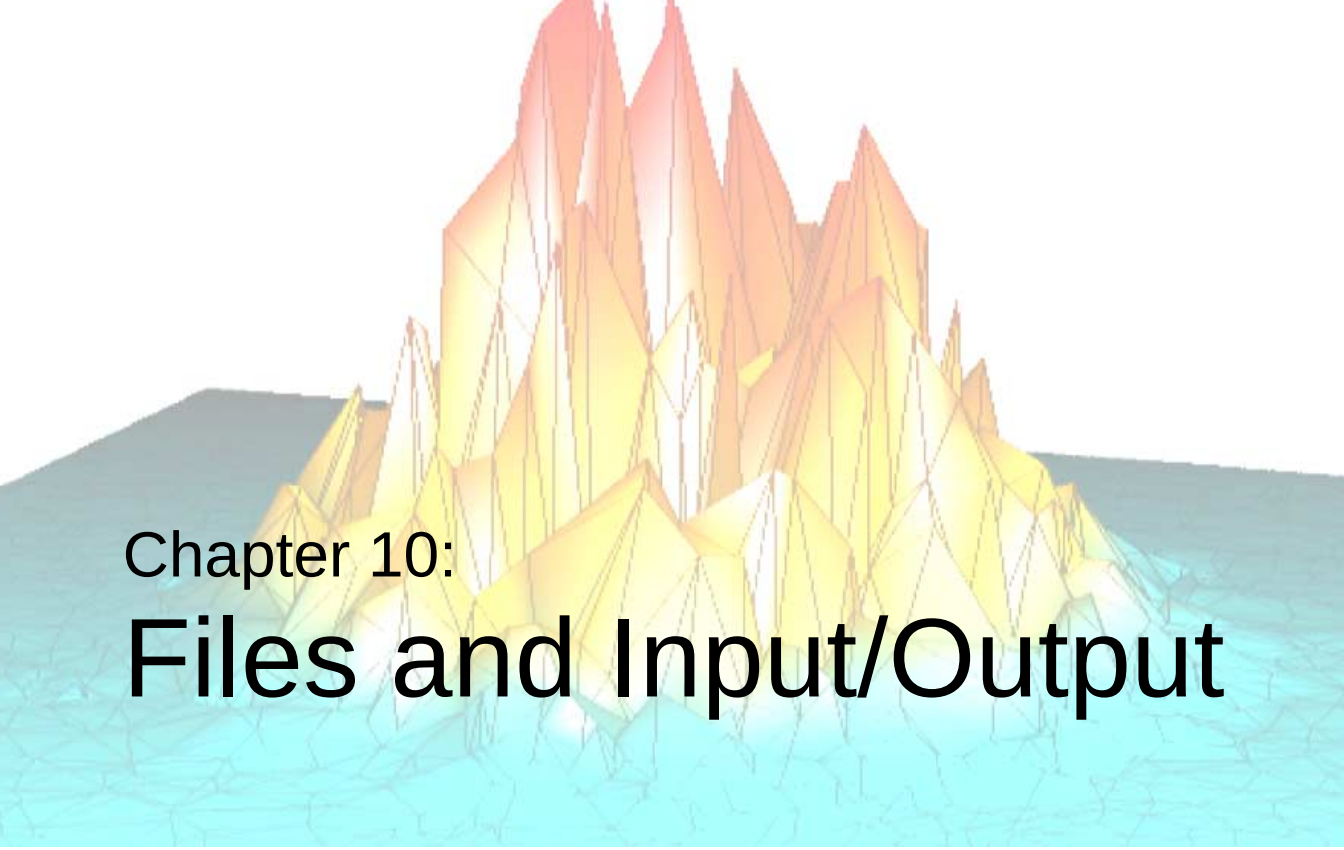
The SAVE command always begins reading any .sav file using 32-bit offsets. If the 64-bit offset command is detected, 64-bit offsets are then used for any subsequent commands.

- 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 `.sav` 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 `.sav` 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 (that is new files being input by old programs). It is not generally reasonable to expect this sort of forward compatibility, and it does not fit the usual definition of backwards compatibility. RSI has always strived to maintain this compatibility. However, in IDL 5.4 this was not the case. The following steps have been taken in IDL 5.5 to minimize the problems that have been caused by the IDL 5.4 save format:

- 64-bit offsets encoding has been improved. The `.sav` files written within IDL 5.5 and subsequently 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 subsequent versions will retain the ability to read the 64-bit offset files produced by IDL 5.4.x, thus ensuring backwards compatibility.
- The `.sav` files written within IDL 5.5 or subsequent versions, which 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 subsequent 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 .sav 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 `RSI-DIR/lib/obsolete`.



Chapter 10:
Files and Input/Output

The following topics are covered in this chapter:

Overview

IDL provides powerful facilities for file input and output. Few restrictions are imposed on data files by IDL, and there is no unique IDL format. This chapter describes IDL input/output methods and routines and gives examples of programs which read and write data using IDL, C, and FORTRAN.

The first section of this chapter provides a description for how IDL input/output works. It is intentionally brief and is intended to serve only as an introduction. Additional details are covered in the following sections. For the IDL user, perhaps the largest single difference between platforms is input/output. The majority of this chapter covers information that is required in all of the environments IDL supports. Operating system specific information is concentrated in the final sections of this chapter.

File I/O in IDL

Before any file input or output can be performed, it is necessary to open a file. This is done using either the OPENR (Open for Reading), OPENW (Open for Writing), or OPENU (Open for Update) procedures. When a file is opened, it is associated with a *Logical Unit Number*, or LUN. All file input and output routines in IDL use the LUN rather than the filename, and most require that the LUN be explicitly specified. Once a file is opened, several input/output routines are available for use. Each routine fills a particular need – the one to use depends on the particular situation.

There are three exceptions to the need to open any file before performing input/output on it. Three files are always open – in fact, the user is not allowed to close them. These files are the *standard input* (usually the keyboard), the *standard output* (usually the IDL log window), and the *standard error output* (usually the terminal screen). These three files are associated with LUNs 0, -1, and -2, respectively. Because these files are always open, there is no need to open them prior to using them for input/output. The READ and PRINT procedures automatically use these files, so basic formatted input/output is extremely simple.

Simple Examples

It is easy to use input/output using the default input and output files. The IDL command:

```
PRINT, 'Hello World.'
```

causes IDL to print the line:

```
Hello World.
```

on the terminal screen. This happens because PRINT formats its arguments and prints them to LUN -1, which is the standard output file. It is only slightly more complicated to use other files. The following IDL statements show how the above “Hello World” example could be sent to a file named *hello.dat*:

```
;Open LUN 1 for hello.dat with write access.
OPENW, 1, 'hello.dat'

;Do the output operation to the file.
PRINTF, 1, 'Hello World.'

;Close the file.
CLOSE, 1
```

Routines for Input/Output

The following routines are useful when doing input/output operations. For more information on these commands, see the *IDL Reference Guide*.

Routine	Description
ASCII_TEMPLATE	Presents a GUI that generates a template defining an ASCII file format.
ASSOC	Associates an array structure with a file.
BINARY_TEMPLATE	Presents a GUI for interactively generating a template structure for use with READ_BINARY.
Alphabetical Listing of CDF Routines	Common Data Format routines.
CLOSE	Closes the specified files.
DIALOG_READ_IMAGE	Presents GUI for reading image files.
DIALOG_WRITE_IMAGE	Presents GUI for writing image files.
EOF	Tests the specified file for the end-of-file condition.
Alphabetical Listing of EOS Routines	HDF-EOS (Hierarchical Data Format-Earth Observing System) routines.
FILEPATH	Returns full path to a file in the IDL distribution.
FINDFILE	Finds all files matching given file specification.
FLUSH	Flushes file unit buffers.
FREE_LUN	Frees previously-reserved file units.
FSTAT	Returns information about a specified file unit.
GET_KBRD	Gets one input IDL character.
GET_LUN	Reserves a logical unit number (file unit).
Alphabetical Listing of HDF Routines	Hierarchical Data Format routines.

Table 10-1: Routines for Input/Output

Routine	Description
Alphabetical Listing of HDF5 Routines	Hierarchical Data Format (version 5) routines.
HDF_BROWSER	Opens GUI to view contents of HDF, HDF-EOS, or NetCDF file.
HDF_READ	Extracts HDF, HDF-EOS, and NetCDF data and metadata into an output structure.
IOCTL	Performs special functions on UNIX files.
MPEG_CLOSE	Closes an MPEG sequence.
MPEG_OPEN	Opens an MPEG sequence.
MPEG_PUT	Inserts an image array into an MPEG sequence.
MPEG_SAVE	Saves an MPEG sequence to a file.
Alphabetical Listing of NCDF Routines	Network Common Data Format routines.
OPEN	Opens files for reading, updating, or writing.
POINT_LUN	Sets or gets current position of the file pointer.
PRINT/PRINTF	Writes formatted output to screen or file.
READ/READF	Reads formatted input from keyboard or file.
READ_ASCII	Reads data from an ASCII file.
READ_BINARY	Reads the contents of a binary file using a passed template or basic command line keywords.
READ_BMP	Reads Microsoft Windows bitmap file (.BMP).
READ_DICOM	Reads an image from a DICOM file.
READ_IMAGE	Reads the image contents of a file and returns the image in an IDL variable.
READ_INTERFILE	Reads Interfile (v3.3) file.
READ_JPEG	Reads JPEG file.

Table 10-1: (Continued) Routines for Input/Output

Routine	Description
READ_PICT	Reads Macintosh PICT (version 2) bitmap file.
READ_PNG	Reads Portable Network Graphics (PNG) file.
READ_PPM	Reads PGM (gray scale) or PPM (portable pixmap for color) file.
READ_SRF	Reads Sun Raster Format file.
READ_SYLK	Reads Symbolic Link format spreadsheet file.
READ_TIFF	Reads TIFF format file.
READ_WAV	Reads the audio stream from the named `.WAV` file.
READ_WAVE	Reads Wavefront Advanced Visualizer file.
READ_X11_BITMAP	Reads X11 bitmap file.
READ_XWD	Reads X Windows Dump file.
READS	Reads formatted input from a string variable.
READU	Reads unformatted binary data from a file.
SOCKET	Opens a client-side TCP/IP Internet socket as an IDL file unit.
TVRD	Reads an image from a window into a variable.
WRITE_BMP	Writes Microsoft Windows Version 3 device independent bitmap file (.BMP).
WRITE_IMAGE	Writes an image and its color table vectors, if any, to a file of a specified type.
WRITE_JPEG	Writes JPEG file.
WRITE_NRIF	Writes NCAR Raster Interchange Format rasterfile.
WRITE_PICT	Writes Macintosh PICT (version 2) bitmap file.
WRITE_PNG	Writes Portable Network Graphics (PNG) file.

Table 10-1: (Continued) Routines for Input/Output

Routine	Description
WRITE_PPM	Writes PPM (true-color) or PGM (gray scale) file.
WRITE_SRF	Writes Sun Raster File (SRF).
WRITE_SYLK	Writes SYLK (Symbolic Link) spreadsheet file.
WRITE_TIFF	Writes TIFF file with 1 to 3 channels.
WRITE_WAV	Writes the audio stream to the named `.WAV` file.
WRITE_WAVE	Writes Wavefront Advanced Visualizer (`.WAV`) file.
WRITEU	Writes unformatted binary data to a file.

Table 10-1: (Continued) Routines for Input/Output

Unformatted Input/Output

Unformatted Input/Output is the most basic form of input/output. Unformatted input/output transfers the internal binary representation of the data directly between memory and the file.

Advantages of Unformatted I/O

Unformatted input/output is the simplest and most efficient form of input/output. It is usually the most compact way to store data.

Disadvantages of Unformatted I/O

Unformatted input/output is the least portable form of input/output. Unformatted data files can only be moved easily to and from computers that share the same internal data representation. It should be noted that XDR (eXternal Data Representation) files, described in “Portable Unformatted Input/Output” on page 272, can be used to produce portable binary data.

Unformatted input/output is not directly human readable, so you cannot type it out on a terminal screen or edit it with a text editor.

Formatted Input/Output

Formatted output converts the internal binary representation of the data to ASCII characters which are written to the output file. Formatted input reads characters from the input file and converts them to internal form. Formatted I/O can be either “Free” format or “Explicit” format, as described below.

Advantages of Formatted I/O

Formatted input/output is very portable. It is a simple process to move formatted data files to various computers, even computers running different operating systems, as long as they all use the ASCII character set. (ASCII is the American Standard Code for Information Interchange. It is the character set used by almost all current computers, with the notable exception of large IBM mainframes.)

Formatted files are human readable and can be typed to the terminal screen or edited with a text editor.

Disadvantages of Formatted I/O

Formatted input/output is more computationally expensive than unformatted input/output because of the need to convert between internal binary data and ASCII text. Formatted data requires more space than unformatted to represent the same information. Inaccuracies can result when converting data between text and the internal representation.

Free Format I/O

With free format input/output, IDL uses default rules to format the data.

Advantages of Free Format I/O

The user is free of the chore of deciding how the data should be formatted. Free format is extremely simple and easy to use. It provides the ability to handle the majority of formatted input/output needs with a minimum of effort.

Disadvantages of Free Format I/O

The default formats used are not always exactly what is required. In this case, explicit formatting is necessary.

Explicit Format I/O

Explicit format I/O allows you to specify the exact format for input/output.

Advantages of Explicit I/O

Explicit formatting allows a great deal of flexibility in specifying exactly how data will be formatted. Formats are specified using a syntax that is similar to that used in FORTRAN format statements. Scientists and engineers already familiar with FORTRAN will find IDL formats easy to write. Commonly used FORTRAN format codes are supported. In addition, IDL formats have been extended to provide many of the capabilities found in the *scanf ()* and *printf ()* functions commonly found in the C language runtime library.

Disadvantages of Explicit I/O

Using explicitly specified formats requires the user to specify more detail—they are, therefore, more complicated to use than free format.

The type of input/output to use in a given situation is usually determined by considering the advantages and disadvantages of each method as they relate to the problem to be solved. Also, when transferring data to or from other programs or systems, the type of input/output is determined by the application. The following suggestions are intended to give a rough idea of the issues involved, though there are always exceptions:

- Images and large data sets are usually stored and manipulated using unformatted input/output in order to minimize processing overhead. The IDL ASSOC function is often the natural way to access such data.
- Data that need to be human readable should be written using formatted input/output.
- Data that need to be portable should be written using formatted input/output. Another option is to use unformatted XDR files by specifying the XDR keyword with the OPEN procedures. This is especially important if moving between computers with markedly different internal binary data formats. XDR is discussed in “Portable Unformatted Input/Output” on page 272.
- 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).

Opening Files

Before a file can be processed by IDL, it must be opened using one of the procedures described in the following table. All open files are associated with a LUN (Logical Unit Number) within IDL, and all input/output routines refer to files via this number. For example, to open the file named *data.dat* for reading on file unit 1, use the following statement:

```
OPENR, 1, 'data.dat'
```

The OPEN procedures can be used with certain keywords to modify their normal behavior. Some keywords are generally applicable, while others only have effect under a given operating system. Some operating system specific keywords are allowed (and ignored) under other operating systems in order to facilitate writing portable routines.

Procedure	Description
OPENR	Opens an existing file for input only.
OPENW	Opens a new file for input and output. If the named file already exists, its old contents are overwritten.
OPENU	Opens an existing file for input and output.

Table 10-2: IDL File Opening Commands

Platform-Specific Keywords to the OPEN Procedure

Different computers and operating systems perform input/output in different ways. See “OPEN” in the *IDL Reference Guide* manual 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.

Logical Unit Numbers (LUNs)

IDL Logical Unit Numbers (LUNs) fall within the range –2 to 128. Some LUNs are reserved for special functions as described below.

The Standard Input, Output, and Error LUNs

The three LUNs described below have special meanings that are operating system dependent:

UNIX

Logical Unit Numbers 0, -1, and -2 are tied to *stdin*, *stdout*, and *stderr*, respectively. This means that the normal UNIX file redirection and pipe operations work with IDL. For example, the shell command

```
%idl < idl.inp >& idl.out &
```

will cause IDL to execute in the background, reading its input from the file *idl.inp* and writing its output to the file *idl.out*. Any messages sent to stderr are also sent to *idl.out*.

When using the IDL Development Environment (IDLDE), Logical Unit Numbers 0, -1, and -2 are tied to *stdin* (the command line), *stdout* (the log window), and *stderr* (the log window), respectively.

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.

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.

Tip

To see if IDL on your platform supports large files, use the following:

```
PRINT, !VERSION.FILE_OFFSET_BITS
```

IF “64” is returned, the platform supports large files. For more information, see “!VERSION” in the *IDL Reference Guide* manual.

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, '000000ffffffff'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" in the *IDL Reference Guide* manual.

- 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, such HP-UX, can only support such operations on special large file systems, and only if they are mounted using the appropriate mount options. Consult your system documentation to determine the limitations present on your system and the procedures for supporting very large file.

Using Free Format Input/Output

Use of formatted data is most appropriate when the data must be in human readable form, such as when it is to be prepared or modified with a text editor. Formatted data also are highly portable between various computers and operating systems.

In addition to the PRINT, PRINTF, READ, and READF routines already discussed, the STRING function can be used to generate formatted output that is sent to a string variable instead of a file. The READS procedure can be used to read formatted input from a string variable.

The exact format of the character data may be specified to these routines by providing a format string via the FORMAT keyword. If no format string is given, default formats for each type of data are applied. This method of formatted input/output is called free format. Free format input/output is suitable for most applications involving formatted data. It is designed to provide input/output abilities with a minimum of programming.

Structures and Free Format Input/Output

IDL structures present a special problem for default formatted input and output. The default format for displaying structure data is to surround the structure with curly braces ({}). For example, if you define an anonymous structure:

```
struct = { A:2, B:3, C:'A String' }
```

and then use default formatted output via the PRINT command:

```
PRINT, struct
```

IDL prints:

```
{       2       3 A String}
```

You might suppose that default formatted input would recognize that the curly braces are part of the formatting and ignore them. This is not the case, however. By default, to read the third field in the structure (the string field) IDL will read from the “A” to the end of the line, including the closing brace.

This behavior, while unsymmetric, seems to be the best choice for default behavior—displaying the result of the PRINT statement on the computer screen. We recommend that you use explicitly formatted input/output when reading and writing structures to disk files, so as not to have to explicitly code around the possibility that your structure may include strings.

Free Format Input

The following rules are used by IDL to perform free format input:

1. Input is performed on scalar variables. Array and structure variables are treated as collections of scalar variables. For example,

   ```
   A = INTARR(5)
   READ, A
   ```

   causes IDL to read five separate values to fill each element of the variable A.

2. If the current input line is empty and there are variables left requiring input, read another line.

3. If the current input line is not empty but there are no variables left requiring input, the remainder of the line is ignored.

4. Input data must be separated by commas or white space (tabs, spaces, or new lines).

5. When reading into a variable of type string, all characters remaining in the current input line are placed into the string.

6. When reading into numeric variables, every effort is made to convert the input into a value of the expected type. Decimal points are optional and exponential (scientific) notation is allowed. If a floating-point datum is provided for an integer variable, the value is truncated.

7. When reading into a variable of complex type, the real and imaginary parts are separated by a comma and surrounded by parentheses. If only a single value is provided, it is taken as the real part of the variable, and the imaginary part is set to zero. For example:

```
;Create a complex variable.
A = COMPLEX(0)

;IDL prompts for input with a colon:
READ, A

;The user enters "(3,4)" and A is set to COMPLEX(3, 4).
:(3, 4)

;IDL prompts for input with a colon:
READ, A

;The user enters "50" and A is set to COMPLEX(50, 0).
:50
```

Free Format Output

The following rules are used by IDL to perform free format output:

1. The format used to output numeric data is determined by the data type. The formats used are summarized in the table below. The formats are specified in the FORTRAN-like style used by IDL for explicitly formatted input/output.

Data Type	Format
Byte	I4
Int, UInt	I8
Long, ULong	I12
Float	G13.6
Long64, ULong64	I22
Double	G16.8
Complex	'(', G13.6, ',', G13.6, ')'
Double-precision Complex	'(', G16.8, ',', G16.8, ')'
String	Output full string on current line.

Table 10-3: Formats Used for Free-Format Output

2. The current output line is filled with characters until one of the following happens:
   A. There is no more data to output.
   B. The output line is full. When output is to a file, the default line width is 80 columns (you can override this default by setting the WIDTH keyword to the OPEN procedure). When the output is to the standard output, IDL uses the current width of your tty or command log window.
   C. An entire row is output in the case of multidimensional arrays.
3. When outputting a structure variable, its contents are bracketed with “{” and “}” characters.

Example: Free Format Input/Output

IDL free format input/output is extremely easy to use. The following IDL statements demonstrate how to read into a complicated structure variable and then print the results:

```
;Create a structure named "types" that contains seven of the basic
;IDL data types, as well as a floating-point array.
A = {TYPES, A:0B, B:0, C:0L, D:1.0, E:1D, $
  F:COMPLEX(0), G: 'string', E:FLTARR(5)}

;Read free-formatted data from input
READ, A

;IDL prompts for input with a colon. We enter values for the first
;six numeric fields of A and the string.
: 1 2 3 4 5 (6,7) EIGHT
```

Notice that the complex value was specified as (6, 7). If the parentheses had been omitted, the complex field of A would have received the value COMPLEX(6, 0), and the 7 would have been input for the next field. When reading into a string variable, IDL starts from the current point in the input and continues to the end of the line. Thus, we do not enter values intended for the rest of the structure on this line.

```
;There are still fields of A that have not received data, so IDL
;prompts for another line of input.
: 9 10 11 12 13

;Show the result.
PRINT, A
```

Executing these statements results in the following output:

```
{   1       2           3      4.00000       5.0000000
(      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 syntax of IDL format strings is extremely similar to that used in FORTRAN. The format string specifies the format in which data is to be transferred as well as the data conversion required to achieve that format. The format specification strings supplied by the FORMAT keyword have the form:

```
FORMAT = '(q1f1s1f2s2 ... fnqn)'
```

where q, f, and s are described below.

Record Terminators

q is zero or more slash (/) record terminators. On output, each record terminator causes the output to move to a new line. On input, each record terminator causes the next line of input to be read.

Format Codes

f is a format code. Some format codes specify how data should be transferred while others control some other function related to how input/output is handled. The code f can also be a nested format specification enclosed in parentheses. This is called a *group specification* and has the following form:

```
...[n](q1f1s1f2s2 ... fnqn) ...
```

A group specification consists of an optional repeat count n followed by a format specification enclosed in parentheses. Use of group specifications allows more compact format specifications to be written. For example, the format specification:

```
FORMAT = '("Result: ", "<",I5,">", "<",I5,">")'
```

can be written more concisely using a group specification:

```
FORMAT = '("Result: ", 2("<",I5,">"))'
```

If the repeat count is 1 or is not given, the parentheses serve only to group format codes for use in format reversion (discussed in the next section).

Field Separators

s is a field separator. A field separator consists of one or more commas (,) and/or slash record terminators (/). The only restriction is that two commas cannot occur side-by-side.

The arguments provided in a call to a formatted input/output routine are called the *argument list*. The argument list specifies the data to be moved between memory and the file. All data are handled in terms of basic IDL components. Thus, an array is considered to be a collection of scalar data elements, and a structure is processed in terms of its basic components. Complex scalar values are treated as two floating-point values.

Rules for Explicitly Formatted Input/Output

IDL uses the following rules to process explicitly formatted input/output:

1. Traverse the format string from left to right, processing each record terminator and format code until an error occurs or no data is left in the argument list. The comma field separator serves no purpose except to delimit the format codes.
2. It is an error to specify an argument list with a format string that does not contain a format code that transfers data to or from the argument list because an infinite loop would result.
3. When a slash record terminator (/) is encountered, the current record is completed, and a new one is started. For output, this means that a new line is started. For input, it means that the rest of the current input record is ignored, and the next input record is read.
4. When a format code that does not transfer data to or from the argument list is encountered, process it according to its meaning. The format codes that do not

transfer data to or from the argument list are summarized in the following table:

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.
nH	FORTRAN-style Hollerith string. Hollerith strings are treated exactly like quoted strings.
nX	Skips *n* character positions.
Tn	Tab. Sets the character position of the next item in the current record.
TLn	Tab Left. Specifies that the next character to be transferred to or from the current record is the *n*-th character to the left of the current position.
TRn	Tab Right. Specifies that the next character to be transferred to or from the current record is the *n*-th character to the right of the current position.

Table 10-4: Format Codes that do not Transfer Data

5. When a format code that transfers data to or from the argument list is encountered, it is matched up with the next datum in the argument list. The

format codes that transfer data to or from the argument list are summarized in the following table:

Code	Action
A	Transfer character data.
C()	Transfer calendar (Julian date and/or time) data.
D	Transfer double-precision, floating-point data.
E	Transfer floating-point data using scientific (exponential) notation.
F	Transfer floating-point data.
G	Use F or E format depending on the magnitude of the value being processed.
I	Transfer integer data.
O	Transfer octal data.
Q	Obtain the number of characters in the input record remaining to be transferred during a read operation. In an output statement, the Q format code has no effect except that the corresponding input/output list element is skipped.
Z	Transfer Hexadecimal data.

Table 10-5: Format Codes that Transfer Data

6. On input, read data from the file and format it according to the format code. If the data type of the input data does not agree with the data type of the variable that is to receive the result, do the type conversion if possible; otherwise, issue a type conversion error and stop.
7. On output, write the data according to the format code. If the data type does not agree with the format code, do the type conversion prior to doing the output if possible. If the type conversion is not possible, issue a type conversion error and stop.
8. If the last closing parenthesis of the format string is reached and there are no data left on the argument list, then format processing terminates. If, however, there are still data to be processed on the argument list, then part or all of the format specification is reused. This process is called *format reversion*.

Format Reversion

In format reversion, the current record is terminated, a new one is initiated, and format control reverts to the group repeat specification whose opening parenthesis matches the next-to-last closing parenthesis of the format string. If the format does not contain a group repeat specification, format control returns to the initial opening parenthesis of the format string. For example, the IDL command:

```
PRINT, FORMAT = '("The values are: ", 2("<", I1, ">"))', $
   INDGEN(6)
```

results in the output

```
The values are: <0><1>
<2><3>
<4><5>
```

The process involved in generating this output is as follows:

1. Output the string “The values are: ”.
2. Process the group specification and output the first two values. The end of the format specification is encountered, so end the output record. Data are remaining, so move back to the group specification
   `2("<", I1, ">")`
   by format reversion.
3. Repeat Step 2 until no data remain. End the output record. Format processing is complete.

Format Codes

“A” Format Code

The A format code transfers character data. The format is

```
[n]A[w]
```

where:

n — is an optional repeat count ($1 \leq n$) specifying the number of times the format code should be processed. If *n* is not specified, a repeat count of one is used.

w — is an optional width ($1 \leq w$) specifying the number of characters to be transferred. If *w* is not specified, the entire string is transferred. On output, if *w* is greater than the length of the string, the string is right justified. On input, IDL strings have dynamic length, so *w* specifies the resulting length of input string variables.

For example, the IDL statement,

```
PRINT, FORMAT = '(A6)', '123456789'
```

generates the following output:

```
123456
```

Note

While an IDL string variable can hold up to 2.1 Gbytes 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.

“:” Format Code

The colon format code terminates format processing if there are no more data remaining in the argument list. For example, the IDL statement,

```
PRINT, FORMAT = '(6(I1, :, ", "))', INDGEN(6)
```

will output the following comma-separated list of integer values:

```
0, 1, 2, 3, 4, 5
```

The use of the colon format code prevented a comma from being output following the final item in the argument list.

“$” Format Code

When IDL completes output format processing, it normally outputs a newline to terminate the output operation. However, if a “$” format code is found in the format specification, this default newline is not output. The “$” format code is only used on output; it is ignored during input formatting. The most common use for the “$” format code is in prompting for user input. For example, the IDL statements,

```
;Prompt for input. Suppress the carriage return.
PRINT, FORMAT = '($, "Enter value: ")'

;Read the response.
READ, VALUE
```

will prompt for input without forcing the user’s response to appear on a separate line from the prompt.

“F,” “D,” “E,” and “G” Format Codes

The F, D, E, and G format codes are used to transfer floating-point values between memory and the specified file. The format is

```
[n]F[w.d]
[n]D[w.d]
[n]E[w.d] or [n]E[w.dEe]
[n]G[w.d] or [n]G[w.dEe]
```

where

n — is an optional repeat count ($1 \le n$) specifying the number of times the format code should be processed. If *n* is not specified, a repeat count of 1 is used.

w.d — is an optional width specification ($1 \le w \le 256$, $1 \le d < w$). The variable *w* specifies the number of characters in the external field. For the F, D, and E format codes, *d* specifies the number of positions after the decimal point. For the G format code, *d* specifies the number of significant digits displayed.

e — is an optional width ($1 \le e \le 256$) specifying the width of exponent part of the field. IDL ignores this value—it is allowed for compatibility with FORTRAN.

On input, the F, D, E, and G format codes all transfer *w* characters from the external field and assign them as a real value to the corresponding input/output argument list datum.

The F and D format codes are used to output values using fixed-point notation. The value is rounded to *d* decimal positions and right-justified into an external field that is *w* characters wide. The value of *w* must be large enough to include a minus sign when necessary, at least one digit to the left of the decimal point, the decimal point, and *d* digits to the right of the decimal point. The code D is identical to F (except for its default values for *w* and *d*) and exists in IDL primarily for compatibility with FORTRAN.

The E format code is used for scientific (exponential) notation. The value is rounded to *d* decimal positions and right-justified into an external field that is *w* characters wide. The value of *w* must be large enough to include a minus sign when necessary, at least one digit to the left of the decimal point, the decimal point, *d* digits to the right of the decimal point, a plus or minus sign for the exponent, the character “e” or “E”, and at least two characters for the exponent.

Note

IDL uses a standard I/O function to format numbers and their exponents. As a result, different platforms may print different numbers of exponent digits.

The G format code uses the F output style when reasonable and E for other values, but displays exactly *d* significant digits rather than *d* digits following the decimal point.

On output, if the field provided is not wide enough, it is filled with asterisks (`*`) to indicate the overflow condition. If *w* is zero, the “natural” width for the value is used—the value is read or output using a default format without any leading or trailing whitespace, in the style of the C standard input/output library `printf (3S)` function. See “C printf-Style Quoted String Format Code” on page 255 for more information on C `printf`-style formatting.

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 10-6: Floating Format Defaults

Using a value of zero for the *w* parameter is useful when reading tables of data in which individual elements may be of varying lengths. For example, if your data reside in tables of the following form:

```
26.01 92.555 344.2
101.0 6.123 99.845
23.723 200.02 141.93
```

setting the format to

```
FORMAT = '(3F0)'
```

ensures that the correct number of digits are read or output for each element.

Normally, the case of the format code is ignored by IDL. However, the case of the E and G format codes determines the case used to output the exponent in scientific notation. The following table gives examples of several floating-point formats and the resulting output.

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 1.0000e+002 (Windows) Note that “e10.4” would not work on Windows because the extra “0” added after the “e” makes the string longer than 10 characters.
E11.4	100.0	*b*1.0000E+02 1.0000E+002 (Windows)
g10.4	100.0	*bbbbb*100.0

Table 10-7: Floating-Point Output Examples (“b” represents a blank space)

Format	Internal Value	Formatted Output
g10.4	10000000.0	*b*1.000e+07 1.000e+007 (Windows)
G10.4	10000000.0	*b*1.000E+07 1.000E+007 (Windows)

Table 10-7: (Continued) Floating-Point Output Examples (“b” represents a

“I,” “O,” and “Z” Format Codes

The I, O, and Z format codes are used to transfer integer values to and from the specified file. The I format code is used to output decimal values, O is used for octal values, and Z is used for hexadecimal values.

The format is as follows:

[*n*]I[*w*] or [*n*]I[*w.m*]
[*n*]O[*w*] or [*n*]O[*w.m*]
[*n*]Z[*w*] or [*n*]Z[*w.m*]

where

n — is an optional repeat count ($1 \le n$) specifying the number of times the format code should be processed. If *n* is not specified, a repeat count of 1 is used.

w — is an optional integer value ($1 \le w \le 256$) specifying the width of the field in characters. The default values used if *w* is omitted are specified in the following table:

Data Type	*w*
Byte, Int, UInt	7
Long, ULong, Float	12
Long64, ULong64	22
Double	23
All Other Types	12

Table 10-8: Integer Format Defaults

If the field provided is not wide enough, it is filled with asterisks (*) to indicate the overflow condition. If *w* is zero, the “natural” width for the value is used—the value

is read or output using a default format without any leading or trailing white space, in the style of the C standard input/output library `printf(3S)` function. See "C printf-Style Quoted String Format Code" on page 255 for more information on C `printf`-style formatting.

Note that using a value of zero for the *w* parameter is useful when reading tables of data in which individual elements may be of varying lengths. For example, if your data reside in tables of the following form:

```
26 92 344
101 6 99
23 200 141
```

setting the format to

```
FORMAT = '(3I0)'
```

ensures that the correct number of digits are read or output for each element.

m — On output, *m* specifies the minimum number of nonblank digits required ($1 \le m \le 256$). The field is zero-filled on the left if necessary. If *m* is omitted or zero, the external field is blank filled.

Normally, the case of the format code is ignored by IDL. However, the case of the Z format codes determines the case used to output the hexadecimal digits A-F. The following table gives examples of several integer formats and the resulting output.

Format	Internal Value	Formatted Output
I	3000	*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

Table 10-9: Integer Output Examples ("b" represents a blank space)

Format	Internal Value	Formatted Output
Z	3000	*bbbb*BB8
Z6.5	3000	*b*00bb8
Z5.6	3000	*****
Z2	3000	**

Table 10-9: (Continued) Integer Output Examples (“b” represents a blank

“Q” Format Code

The Q format code returns the number of characters in the input record remaining to be transferred during the current read operation. It is ignored during output formatting. Format Q is useful for determining how many characters have been read on a line. For example, the following IDL statements count the number of characters in file *demo.dat*:

```
;Open file for reading.
OPENR, 1, "demo.dat"

;Create a longword integer to keep the count.
N = 0L

;Count the characters.
WHILE(~ EOF(1)) DO BEGIN
  READF, 1, CUR, FORMAT = '(q)' & N = N + CUR
END

;Report the result.
PRINT, FORMAT = '("counted", N, "characters.")'

;Close file.
CLOSE, 1
```

Quoted String and “H” Format Codes

On output, any quoted strings or Hollerith constants are sent directly to the output. On input, they are ignored. For example, the IDL statement,

```
PRINT, FORMAT = '("Value: ", I0)', 23
```

results in the following output:

```
Value: 23
```

Notice the use of single quotes around the entire format string and double quotes around the quoted string inside the format. This is necessary because we are including quotes inside a quoted string. It would have been equally correct to use double quotes around the entire format string and single quotes internally. Another way to specify the string is with a Hollerith constant as follows:

```
PRINT, FORMAT = '(7HValue: , I0)', 23
```

The format for a Hollerith constant is:

*n*Hc_1c_2 c_3 ... *cn*

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*.

See “C printf-Style Quoted String Format Code” on page 255 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 format is

T*n*

where

n — is the absolute character position within the record to which the current position should be set ($1 \le n$).

T — differs from the TL, TR, and X format codes primarily in that it requires an absolute position rather than an offset from the current position. For example,

```
PRINT, FORMAT = '("First", 20X, "Last", T10, "Middle")'
```

produces the following output:

```
First    Middle          Last
```

where “*b*” represents a blank space.

"TL" Format Code

The TL format code moves the current position in the external record to the left. The format is

```
TLn
```

where

n — is the number of characters to move left from the current position ($1 \leq n$). If the value of *n* is greater than the current position, the current position is moved to column one.

TL — is used to move backwards in the current record. It can be used on input to read the same data twice or on output to position the output nonsequentially. For example,

```
PRINT, FORMAT = '("First", 20X, "Last", TL15, "Middle")'
```

produces the following output:

```
FirstbbbbbbbbbMiddlebbbbbLast
```

where "*b*" represents a blank space.

"TR" and "X" Format Codes

The TR and X format codes move the current position in the record to the right. The format is

```
TRn
nX
```

where

n — is the number of characters to skip ($1 \leq n$). On input, *n* characters in the current input record will be passed over. On output, the current output position is moved *n* characters to the right.

The TR or X format codes can be used to leave whitespace in the output or to skip over unwanted data in the input. For example,

```
PRINT, FORMAT = '("First", 15X, "Last")'
```

or

```
PRINT, FORMAT = '("First", TR15, "Last")'
```

results in the following output:

```
FirstbbbbbbbbbbbbbbbLast
```

where "*b*" represents a blank space.

These two format codes differ in one way. Using the X format code at the end of an output record will not cause any characters to be written unless it is followed by another format code that causes characters to be output. The TR format code always writes characters in this situation. Thus,

```
PRINT, FORMAT = '("First", 15X)'
```

does not leave 15 blanks at the end of the line, but the following statement does:

```
PRINT, FORMAT = '("First", TR15)'
```

“C()” Format Code

The C() format code is used to transfer calendar (Julian date and/or time) data. The format is

```
[n]C([c0,c1,...,cx])
```

where:

n — is an optional repeat count ($1 \leq n$) specifying the number of times the format code should be processed. If *n* is not specified, a repeat count of 1 is used.

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 $6x10^{-10}$ (which corresponds to $5x10^{-5}$ 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:

"CMOA" subcodes

The CMOA subcodes transfers the month portion of a date as a string. The format for an all upper case month string is:

```
CMOA[w]
```

The format for a capitalized month string is:

```
CMoA[w]
```

The format for an all lower case month string is:

```
CmoA[w]
```

Note

The case of the 'M' and 'O' of these subcodes will be ignored on input, or if the MONTHS keyword for the current routine is explicitly set.

For these subcodes:

w — is an optional width ($0 \le w \le 256$) specifying the number of characters of the month name to be transferred. If *w* is not specified, three characters will be transferred. If *w* is 0, the natural length of the month name is transferred. On output, if *w* is greater than the natural length of the month name, the string will be right justified.

"CMOI" subcode

The CMOI subcode transfers the month portion of a date as an integer. The format is as follows:

```
CMOI[w] or CMOI[w.m]
```

where:

w — is an optional width ($1 \le w \le 256$) specifying the width of the field in characters. The default width is 2.

m — On output, *m* specifies the minimum number of nonblank digits required ($1 \le m \le 256$). The field is zero-filled on the left if necessary. If *m* is omitted or zero, the external field is blank filled.

"CDI" subcode

The CDI subcode transfers the day portion of a date as an integer. The format is as follows:

```
CDI[w] or CDI[w.m]
```

where:

w — is an optional width ($1 \le w \le 256$) specifying the width of the field in characters. The default width is 2.

m — On output, *m* specifies the minimum number of nonblank digits required ($1 \le m \le 256$). The field is zero-filled on the left if necessary. If m is omitted or zero, the external field is blank filled.

"CYI" subcode

The CYI subcode transfers the year portion of a date as an integer. The format is as follows:

```
CYI[w] or CYI[w.m]
```

where:

w — is an optional width ($1 \le w \le 256$) specifying the width of the field in characters. The default width is 4.

m — On output, *m* specifies the minimum number of nonblank digits required ($1 \le m \le 256$). The field is zero-filled on the left if necessary. If *m* is omitted or zero, the external field is blank filled.

"CHI" subcodes

The CHI subcodes transfer the hour portion of a date as an integer. The format for 24 hour based integer is:

```
CHI[w] or CHI[w.m]
```

The format for a 12 hour based integer is:

```
ChI[w] or ChI[w.m]
```

For these subcodes:

w — is an optional width ($1 \le w \le 256$) specifying the width of the field in characters. The default width is 2.

m — On output, *m* specifies the minimum number of nonblank digits required ($1 \le m \le 256$). The field is zero-filled on the left if necessary. If *m* is omitted or zero, the external field is blank filled.

"CMI" subcode

The CMI subcode transfers the minute portion of a date as an integer. The format is as follows:

```
CMI[w] or CMI[w.m]
```

where:

w — is an optional width ($1 \le w \le 256$) specifying the width of the field in characters. The default width is 2.

m — On output, *m* specifies the minimum number of nonblank digits required ($1 \le m \le 256$). The field is zero-filled on the left if necessary. If *m* is omitted or zero, the external field is blank filled.

"CSI" subcode

The CSI subcode transfers the seconds portion of a date as an integer. The format is as follows:

```
CSI[w] or CSI[w.m]
```

where:

w — is an optional width ($1 \le w \le 256$) specifying the width of the field in characters. The default width is 2.

m — On output, *m* specifies the minimum number of nonblank digits required ($1 \le m \le 256$). The field is zero-filled on the left if necessary. If *m* is omitted or zero, the external field is blank filled.

"CSF" subcode

The CSF subcode transfers the seconds portion of a date as a floating-point value. The format is as follows:

```
CSF[w.d]
```

where:

w.d — is an optional width specification ($1 \le w \le 256$, $1 \le d < w$). The variable *w* specifies the number of characters in the external field; the default is 5. The variable *d* specifies the number of positions after the decimal point; the default is 2. The value of *w* must be large enough to include at least one digit to the left of the decimal point, the decimal point, and *d* digits to the right of the decimal point. On output, if the field provided is not wide enough, it is filled with asterisks (*) to indicate the overflow condition. If *w* is zero, the "natural" width for the value is used – the value is read or output using a default format without any leading or trailing whitespace, in the style of the C standard library printf (3S) function.

"CDWA" subcodes

The CDWA subcodes transfers the day of week portion of a data as a string. The format for an all upper case day of week string is:

```
CDWA[w]
```

The format for a capitalized day of week string is:

```
CDwA[w]
```

The format for an all lower case day of week string is:

```
CdwA[w]
```

Note

The case of the 'D' and 'W' of these subcodes will be ignored on input, or if the DAYS_OF_WEEK keyword for the current routine is explicitly set.

For these subcodes:

w — is an optional width ($0 \le w \le 256$), specifying the number of characters of the day of week name to be transferred. If *w* is not specified, three characters will be transferred. If *w* is 0, the natural length of the day of week name is transferred. On output, if *w* is greater than the natural length of the day of week name, the string will be right justified.

"CAPA" subcodes

The CAPA subcodes transfers the am or pm portion of a date as a string. The format for an all upper case AM or PM string is:

```
CAPA[w]
```

The format for a capitalized AM or PM string is:

```
CApA[w]
```

The format for an all lower case AM or PM string is:

```
CapA[w]
```

Note

The case of the first ‘A’ and ‘P’ of these subcodes will be ignored on input, or if the AM_PM keyword for the current routine is explicitly set.

For these subcodes:

w — is an optional width ($0 \le w \le 256$), specifying the number of characters of the AM or PM string to be transferred. If *w* is not specified, two characters will be transferred. If *w* is 0, the natural length of the AM or PM string is transferred. On output, if *w* is greater than the natural length of the AM or PM string, the string will be right justified.

Standard Format Codes Allowed within a Calendar Specification

None of these subcodes are allowed outside of a C() format specifier. In addition to the subcodes listed above, only quoted strings, “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,":",CSF5.2,TL5,CSI2.2))',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 use “TL” (tab left) and then overwrite the integer portion.

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 `sprintf()`. 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 246) 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.

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 241
%[*w*]d or %[*w*]D %[*w.m*]D or %[*w.m*]D %[*w*]i or %[*w*]I %[*w.m*]i or %[*w.m*]I	I[*w*] I[*w.d*] I[*w*] I[*w.d*]	““I,” “O,” and “Z” Format Codes” on page 244
%[*w.d*]f or %[*w.d*]F	F[*w.d*]	““F,” “D,” “E,” and “G” Format Codes” on page 241
%[*w.d*]g or %[*w.d*]G	g[*w.d*] or G[*w.d*]	““F,” “D,” “E,” and “G” Format Codes” on page 241
%[*w*]o or %[*w*]O %[*w.m*]o or %[*w.m*]O	O[*w*] O[*w.d*]	““I,” “O,” and “Z” Format Codes” on page 244
%[*w*]s or %[*w*]S	A[*w*]	““A” Format Code” on page 240

Table 10-10: Supported “%” Formats

Printf-Style	Normal-Style	Normal Style Described in Section
%[*w*]x or %[*w*]X	Z[*w*]	""I," "O," and "Z" Format Codes" on page 244
%[*w.m*]x or %[*w.m*]X	Z[*w.d*]	
%[*w*]z or %[*w*]Z	Z[*w*]	
%[*w.m*]z or %[*w.m*]Z	Z[*w.d*]	

Table 10-10: 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*, 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.

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 \a	BEL (7B)
\B \b	Backspace (8B)
\F \f	Formfeed (12B)
\N \n	Linefeed (10B)
\R \r	Carriage Return (13B)

Table 10-11: Supported "\" Character Escapes

Escape Sequence	ASCII code
\T \t	Horizontal Tab (9B)
\V \v	Vertical Tab (11B)
\ooo	Octal value ooo (Octal value of 1-3 digits)
\xhh	Hexadecimal value xx (Hex value of 1-2 digits)

Table 10-11: (Continued) Supported "\" Character Escapes

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 256 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.

- The C `printf` function allows the use of %-wd notation to specify that the data should be left justified in a field of w characters. IDL does not support this notation.
- 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.

Example: Reading Tables of Formatted Data

IDL explicitly formatted input/output has the power and flexibility to handle almost any kind of formatted data. A common use of explicitly formatted input/output involves reading and writing tables of data. Consider a data file containing employee data records. Each employee has a name (String, 32 columns) and the number of years they have been employed (Integer, 3 columns) on the first line. The next two lines contain each employee's monthly salary for the last twelve months. A sample file named *employee.dat* with this format might look like the following:

```
Bullwinkle                       10
1000.0    9000.97   1100.0                    2000.0
5000.0    3000.0    1000.12   3500.0    6000.0     900.0
Boris                             11
400.0     500.0    1300.10    350.0     745.0    3000.0
200.0     100.0     100.0      50.0      60.0       0.25
Natasha                           10
950.0    1050.0    1350.0     410.0     797.0     200.36
2600.0    2000.0    1500.0    2000.0    1000.0     400.0
Rocky                             11
1000.0    9000.0    1100.0       0.0       0.0    2000.37
5000.0    3000.0    1000.01   3500.0    6000.0     900.12
```

The following IDL statements read data with the above format and produce a summary of the contents of the file:

```
;Open data file for input.
OPENR, 1, 'employee.dat'

;Create variables to hold the name, number of years, and monthly
;salary.
name = '' & years = 0 & salary = FLTARR(12)

;Output a heading for the summary.
PRINT, FORMAT='("Name", 28X, "Years", 4X, "Yearly Salary")'

;Note: The actual dashed line is longer than is shown here.
PRINT, '========'
```

```
;Loop over each employee.
WHILE (~ EOF(1)) DO BEGIN

   ;Read the data on the next employee.
   READF, 1, $
   FORMAT = '(A32,I3,2(/,6F10.2))', name, years, salary

;Output the employee information. Use TOTAL to sum the monthly
;salaries to get the yearly salary.
   PRINT, FORMAT='(A32,I5,5X,F10.2)', name, years, TOTAL(salary)

ENDWHILE

CLOSE, 1
```

The output from executing these statements on *employee.dat* is as follows:

```
Name                            Years    Yearly Salary
=======================================================
Bullwinkle                         10       32501.09
Borris                             11        6805.35
Natasha                            10       14257.36
Rocky                              11       32500.50
```

Example: Reading Records that Contain Multiple Array Elements

Frequently, data are written to files with each record containing single elements of more than one array. One example might be a file consisting of observations of altitude, pressure, temperature, and velocity with each line or record containing a value for each of the four variables. Because IDL has no equivalent of the FORTRAN implied DO list, special procedures must be used to read or write this type of file.

The first approach, which is the simplest, may be used only if all of the variables have the same data type. An array is created with as many columns as there are variables and as many rows as there are elements. The data are read into this array, the array is transposed storing each variable as a row, and each row is extracted and stored into a variable which becomes a vector. For example, the FORTRAN program which writes the data and the IDL program which reads the data are as follows:

FORTRAN Write:

```
DIMENSION ALT(100),PRES(100),TEMP(100),VELO(100)
OPEN (UNIT = 1, STATUS='NEW', FILE='TEST')
WRITE(1,'(4(1x,g15.5))')
     (ALT(I),PRES(I),TEMP(I),VELO(I),I=1,100)
```

IDL Read:

```
;Open file for input.
OPENR, 1, 'test'

;Define variable (NVARS by NOBS).
A = FLTARR(4,100)

;Read the data.
READF, 1, A

;Transpose so that columns become rows.
A = TRANSPOSE(A)

;Extract the variables.
ALT = A[*, 0]
PRES = A[*, 1]
TEMP = A[*, 2]
VELO = A[*, 3]
```

Note that this same example may be written without the implied DO list, writing all elements for each variable contiguously and simplifying matters considerably:

FORTRAN Write:

```
DIMENSION ALT(100),PRES(100),TEMP(100),VELO(100)
OPEN (UNIT = 1, STATUS='NEW', FILE='TEST')
WRITE (1,'(4(1x,G15.5))') ALT,PRES,TEMP,VELO
```

IDL Read:

```
;Define variables.
ALT = FLTARR(100)
PRES = ALT & TEMP = ALT & VELO = ALT
OPENR, 1, 'test'
READF, 1, ALT, PRES, TEMP, VELO
```

A different approach must be taken when the columns contain different data types or the number of lines or records are not known. This method involves defining the arrays, defining a scalar variable to contain each datum in one record, then writing a loop to read each line into the scalars, and then storing the scalar values into each array. For example, assume that a fifth variable, the name of an observer which is of string type, is added to the variable list. The FORTRAN output routine and IDL input routine are as follows:

FORTRAN Write:

```
DIMENSION ALT(100),PRES(100),TEMP(100),VELO(100)
```

```
CHARACTER * 10 OBS(100)
OPEN (UNIT = 1, STATUS = 'NEW', FILE = 'TEST')
WRITE (1,'(4(1X,G15.5),2X,A)')
  (ALT(I),PRES(I),TEMP(I),VELO(I),OBS(I),I=1,100)
```

IDL Read:

```
;Access file. 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 277. The READU and WRITEU procedures provide IDL's basic unformatted input/output capabilities. They have the form:

```
READU, Unit, Var1, ..., Varn
WRITEU, Unit, Var1, ..., Varn
```

where

Unit — The logical file unit with which the input/output operation will be performed.

Var_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.

```
#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:

```
#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 265).
- 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 10-12: 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” in the *IDL Reference Guide* manual 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.

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 265.

File Manipulation Operations

IDL provides a variety of routines that allow you to retrieve information about and manipulate files and directories.

IDL File Handling Routines

IDL file handling routines are listed in the following table:

File Handling Routine	Description
FILE_CHMOD	Allows you to change file access permissions.
FILE_COPY	Allows you to copy files and directories.
FILE_DELETE	Allows you to delete files and empty directories.
FILE_EXPAND_PATH	Fully qualifies file and directory paths.
FILE_INFO	Returns file status information.
FILE_LINES	Returns the number of lines of text in a specified file.
FILE_LINK	Creates UNIX links.
FILE_MKDIR	Creates directories.
FILE_MOVE	Allows you to rename files and directories.
FILE_READLINK	Returns the path to a file referenced by a UNIX symbolic link.
FILE_SAME	Allows you to determine whether two file names refer to the same file.
FILE_SEARCH	Finds files whose names match a specified string.
FILE_TEST	Tests a file or directory for existence and other specific attributes.
FILE_WHICH	Searches for a specified file in a directory search path.

Table 10-13: IDL File Handling Routines

Locating Files

The FILE_SEARCH function returns an array of strings containing the names of all files that match its argument string. The argument string may contain any wildcard characters understood by the command interpreter. For example, to determine the number of IDL procedure files that exist in the current directory, use the following statement:

```
PRINT, '# IDL pro files:',N_ELEMENTS(FILE_SEARCH('*.pro'))
```

See "FILE_SEARCH" in the *IDL Reference Guide* manual for details.

Changing File Access Permissions

The FILE_CHMOD procedure allows you to change the current access permissions (also referred to as modes) associated with a file or directory. File modes are specified using the standard Posix convention of three protection classes (user, group, other), each containing three attributes (read, write, execute). This is the same format familiar to users of the UNIX `chmod(1)` command. For example, to make the file `moose.dat` read-only to everyone except the owner of the file, but otherwise not change any other settings:

```
FILE_CHMOD, 'moose.dat', /u_write, g_write=0, o_write=0
```

To make the file be readable and writable to the owner and group, but read-only to anyone else, and remove any other modes:

```
FILE_CHMOD, 'moose.dat', '664'
```

To find the current protection settings for a given file, you can use the GET_MODE keyword to the FILE_TEST function.

See "FILE_CHMOD" in the *IDL Reference Guide* manual for details.

Copying Files and Directories

The FILE_COPY procedure allows you to copy files and directories from one location to another. The copies retain the protection settings of the original files, and belong to the user that performed the copy.

See "FILE_COPY" in the *IDL Reference Guide* manual for details.

Renaming Files and Directories

The FILE_MOVE procedure allows you to rename files and directories. The moved files retain their protection and ownership attributes. Within a given file system or

volume, FILE_MOVE does not copy file data. Rather, it simply changes the file names by updating the directory structure of the file system.

See "FILE_MOVE" in the *IDL Reference Guide* manual for details.

Deleting Files and Empty Directories

The FILE_DELETE procedure allows you to delete files and empty directories for which they have appropriate permission. The process must have the necessary permissions to remove the file, as defined by the current operating system. FILE_CHMOD can be used to change file protection settings.

Microsoft Windows users should be careful to not specify a trailing backslash at the end of a specification. For example:

```
FILE_DELETE, 'c:\mydir\myfile'
```

and not:

```
FILE_DELETE, 'c:\mydir\myfile\'
```

See "FILE_DELETE" in the *IDL Reference Guide* manual for details.

Expanding Files and Directory Paths

The FILE_EXPAND_PATH function can be used with a given a file or directory name to convert the name to its fully qualified form and return it. A fully-qualified file path completely specifies the location of a file without the need to consider the user's current working directory.

Note

This routine should be used only to make sure that file paths are fully qualified, but not to expand wildcard characters (e.g. *). The behavior of FILE_EXPAND_PATH when it encounters a wildcard is platform dependent, differs between platforms, and should not depended on. These differences are due to the underlying operating system, and are beyond the control of IDL. To expand the wildcard and obtain fully qualified paths, combine the FILE_SEARCH function with FILE_EXPAND_PATH:

```
A = FILE_EXPAND_PATH(FILE_SEARCH('*.pro'))
```

Alternatively, you can use the FULLY_QUALIFY_PATH keyword to FILE_SEARCH:

```
A = FILE_SEARCH('*.pro', /FULLY_QUALIFY_PATH)
```

See "FILE_EXPAND_PATH" in the *IDL Reference Guide* manual for details.

Creating Directories

You can create a directory using the FILE_MKDIR procedure. The resulting directory or directories are created with default access permissions for the current process. If needed, you can use the FILE_CHMOD procedure to alter access permissions. If a specified directory has non-existent parent directories, FILE_MKDIR automatically creates all the intermediate directories as well. For instance, to create a subdirectory named `moose` in the current working directory:

```
FILE_MKDIR, 'moose'
```

See "FILE_MKDIR" in the *IDL Reference Guide* manual for details.

Testing for a File's Existence

The FILE_TEST function allows you to determine if a file exists without having to open it. Additionally, using the FILE_TEST keywords provides information about the file's attributes. For example, to determine whether your IDL distribution supports the SGI IRIX operating system:

```
result = FILE_TEST(!DIR + '/bin/bin.sgi', /DIRECTORY)
PRINT, 'SGI IDL Installed: ', result ? 'yes' : 'no'
```

See "FILE_TEST" in the *IDL Reference Guide* manual for details.

Searching for a Specific File

The FILE_WHICH function separates a specified file path into its component directories, and searches each directory in turn. If a directory contains the file, the full name of that file including the directory path is returned. If FILE_WHICH does not find the desired file, a NULL string is returned.

This command is modeled after the UNIX `which(1)` command, but is written in the IDL language and is available on all platforms. Its source code can be found in the file `file_which.pro` in the `lib` subdirectory of the IDL distribution.

As an example, the following line of code allows you to find the location of the `file_which.pro` file:

```
Result = FILE_WHICH('file_which.pro')
```

Alternately, to find the location of the UNIX `ls` command:

```
Result = FILE_WHICH(getenv('PATH'), 'ls')
```

See "FILE_WHICH" in the *IDL Reference Guide* manual for details.

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" in the *IDL Reference Guide* manual for details.

Use the FILE_READLINK procedure to retrieve the path to a file referenced by a UNIX symbolic link. See "FILE_READLINK" in the *IDL Reference Guide* manual for details.

Use the FILE_SAME function to determine whether two file names refer to the same underlying file. See "FILE_SAME" in the *IDL Reference Guide* manual for details.

Getting Help and Information

Information about currently open file units is available by using the FILES keyword with the HELP procedure. If no arguments are provided, information about all currently open user file units (units 1–128) is given. For example, the following command can be used to get information about the three special units (–2, –1, and 0):

```
HELP, /FILES, -2, -1, 0
```

This command results in output similar to the following:

```
Unit    Attributes                       Name
-2      Write, New, Tty, Reserved       <stderr>
```

```
-1      Write, New, Tty, Reserved        <stdout>
 0      Read, Tty, Reserved              <stdin>
```

See "HELP" in the *IDL Reference Guide* manual for details.

Getting Information About a File

You can use the FILE_INFO function to retrieve information about a file that is not currently open (that is, for which there is no IDL Logical Unit Number available). To get information about an open file, use the FSTAT function.

The FILE_INFO function returns a structure expression of type FILE_INFO containing information about the file. For example, use to get information on the file `dist.pro` within the IDL User Library:

```
HELP,/STRUCTURE, FILE_INFO(FILEPATH('dist.pro', $
    SUBDIRECTORY = 'lib'))
```

The above command will produce output similar to:

```
** Structure FILE_INFO, 21 tags, length=72:
        NAME            STRING    '/usr/local/rsi/idl/lib/dist.pro'
        EXISTS          BYTE         1
        READ            BYTE         1
        WRITE           BYTE         0
        EXECUTE         BYTE         0
        REGULAR         BYTE         1
        DIRECTORY       BYTE         0
        BLOCK_SPECIAL   BYTE         0
        CHARACTER_SPECIAL
                        BYTE         0
        NAMED_PIPE      BYTE         0
        SETGID          BYTE         0
        SETUID          BYTE         0
        SOCKET          BYTE         0
        STICKY_BIT      BYTE         0
        SYMLINK         BYTE         0
        DANGLING_SYMLINK
                        BYTE         0
        MODE            LONG               420
        ATIME           LONG64                 970241431
        CTIME           LONG64                 970241595
        MTIME           LONG64                 969980845
        SIZE            LONG64                      1717
```

The fields of the FILE_INFO structure 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 FILE_INFO structure, see "FILE_INFO" in the *IDL Reference Guide* manual.

Use the FILE_LINES function to retrieve the number of lines of text in a file. See "FILE_LINES" in the *IDL Reference Guide* manual for details.

The FSTAT Function

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" in the *IDL Reference Guide* manual.

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)
```

Flushing File Units

For efficiency, IDL buffers its input/output in memory. Therefore, when data are output, there is a window of time during which data are in memory and have not been

actually placed into the file. Normally, this behavior is transparent to the user (except for the improved performance). The FLUSH routine exists for those rare occasions where a program needs to be certain that the data has actually been written to the file immediately. For example, use the statement,

```
FLUSH, 1
```

to flush file unit one.

See "FLUSH" in the *IDL Reference Guide* manual 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" in the *IDL Reference Guide* manual 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" in the *IDL Reference Guide* manual for details.

GET_KBRD

The GET_KBRD function returns the next character available from the standard input (IDL file unit zero) as a single character string. It takes a single parameter named WAIT. If WAIT is zero, the function returns the null string if there are no characters in the terminal typeahead buffer. If it is nonzero, the function waits for a character to be typed before returning.

Under Windows, the GET_KBRD function can be used to return Windows special characters (in addition to the standard keyboard characters). To get a special character, hold down the Alt key and type the character's ANSI equivalent on the numeric keypad while GET_KBRD is waiting. Control + *key* combinations are not supported.

See "GET_KBRD" in the *IDL Reference Guide* manual for details.

Note

RSI recommends the use of a GUI interface (e.g. WIDGET_BUTTON) instead of GET_KBRD where possible.

Example—Using GET_KBRD

A procedure that updates the screen and exits when the carriage return is typed might appear as follows:

```
;Procedure definition.
PRO UPDATE, ...

;Loop forever.
WHILE 1 DO BEGIN

;Update screen here...
...
```

```
;Read character, no wait.
CASE GET_KBRD(0) OF

;Process letter A.
'A': ....

;Process letter B.
'B': ....

;Process other alternatives.
...

;Exit on carriage return (ASCII code = 15 octal).
STRING("15B): RETURN

;Ignore all other characters.
ELSE:

ENDCASE

ENDWHILE

;End of procedure.
END
```

Using the STRING Function to Format Data

The STRING function is very similar to the PRINT and PRINTF procedures. It can be thought of as a version of PRINT that places its formatted output into a string variable instead of a file. If the output is a single line, the result is a scalar string. If the output has multiple lines, the result is a string array with each element of the array containing a single line of the output.

Example—Using STRING with Explicit Formatting

The IDL statements:

```
;Produce a string array.
A=STRING(FORMAT='("The values are:", /, (I))', INDGEN(5))

;Show its structure.
HELP, A

;Print out the result.
FOR I = 0, 5 DO PRINT, A[I]
```

produce the following output:

```
A               STRING    = Array(6)
The values are:
      0
      1
      2
      3
      4
```

See “STRING” in the *IDL Reference Guide* manual for details.

Reading Data from a String Variable

The READS procedure performs formatted input from a string variable and writes the results into one or more output variables. This procedure differs from the READ procedure only in that the input comes from memory instead of a file.

This routine is useful when you need to examine the format of a data file before reading the information it contains. Each line of the file can be read into a string using READF. Then the components of that line can be read into variables using READS.

See the description of “READS” in the *IDL Reference Guide* manual for more details.

UNIX-Specific Information

UNIX offers only a single type of file. All files are considered to be an uninterpreted stream of bytes, and there is no such thing as record structure at the operating system level. (By convention, records of text are simply terminated by the linefeed character, which is referred to as “newline.”) It is possible to move the current file pointer to any arbitrary position in the file and to begin reading or writing data at that point. This simplicity and generality form a system in which any type of file can be manipulated easily using a small set of file operations.

Reading FORTRAN-Generated Unformatted Data with IDL

The UNIX file system considers all files to be an uninterpreted stream of bytes. Standard FORTRAN I/O considers all input/output to be done in terms of logical records.

In order to reconcile the FORTRAN need for logical records with UNIX files, UNIX FORTRAN programs add a longword count before and after each logical record of data. These longwords contain an integer count giving the number of bytes in that record. Note that direct-access FORTRAN I/O does not write data in this format, but simply transfers binary data to or from the file.

The use of the F77_UNFORMATTED keyword with the OPENR statement informs IDL that the file contains unformatted data produced by a UNIX FORTRAN program. When a file is opened with this keyword, IDL interprets the longword counts properly and is able to read and write files that are compatible with FORTRAN.

Reading data from a FORTRAN file

The following UNIX FORTRAN program produces a file containing a five-column by three-row array of floating-point values with each element set to its one-dimensional subscript:

```
PROGRAM ftn2idl

INTEGER i, j
REAL data(5, 3)

OPEN(1, FILE="ftn2idl.dat", FORM="unformatted")
  DO 100 j = 1, 3
    DO 100 i = 1, 5
      data(i,j) = ((j - 1) * 5) + (i - 1)
```

```
          print *, data(i,j)
100 CONTINUE
    WRITE(1) data
    END
```

Running this program creates the file *ftn2idl.dat* containing the unformatted array. The following IDL statements can be used to read this file and print out its contents:

```
;Create an array to contain the fortran array.
data = FLTARR(5,3)

;Open the fortran-generated file. The F77_UNFORMATTED keyword is
;necessary so that IDL will know that the file contains unformatted
;data produced by a UNIX FORTRAN program.
OPENR, lun, 'ftn2idl.dat', /GET_LUN, /F77_UNFORMATTED

;Read the data in a single input operation.
READU, lun, data

;Release the logical unit number and close the fortran file.
FREE_LUN, lun

;Print the result.
PRINT, data
```

Executing these IDL statements produces the following output:

```
0.00000      1.00000      2.00000      3.00000      4.00000
5.00000      6.00000      7.00000      8.00000      9.00000
10.0000      11.0000      12.0000      13.0000      14.0000
```

Because unformatted data produced by UNIX FORTRAN unformatted WRITE statements are interspersed with extra information before and after each logical record, it is important that the IDL program read the data in the same way that the FORTRAN program wrote it. For example, consider the following attempt to read the above data file one row at a time:

```
;Create an array to contain one row of the FORTRAN array.
data = FLTARR(5, /NOZERO)

OPENR, lun, 'ftn2idl.dat', /GET_LUN, /F77_UNFORMATTED

;One row at a time.
FOR I = 0, 4 DO BEGIN

   ;Read a row of data.
   READU, lun, data

   ;Print the row.
   PRINT, data
```

```
ENDFOR

;Close the file.
FREE_LUN, lun
```

Executing these IDL statements produces the output:

```
0.00000      1.00000      2.00000      3.00000      4.00000
% READU: End of file encountered. Unit: 100
         File: ftn2idl.dat6
% Execution halted at $MAIN$(0).
```

Here, IDL attempted to read the single logical record written by the FORTRAN program as if it were written in five separate records. IDL hit the end of the file after reading the first five values of the first record.

Writing data to a FORTRAN file

The following IDL statements create a five-column by three-row array of floating-point values with each element set to it's 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
```

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.

Scientific Data Formats

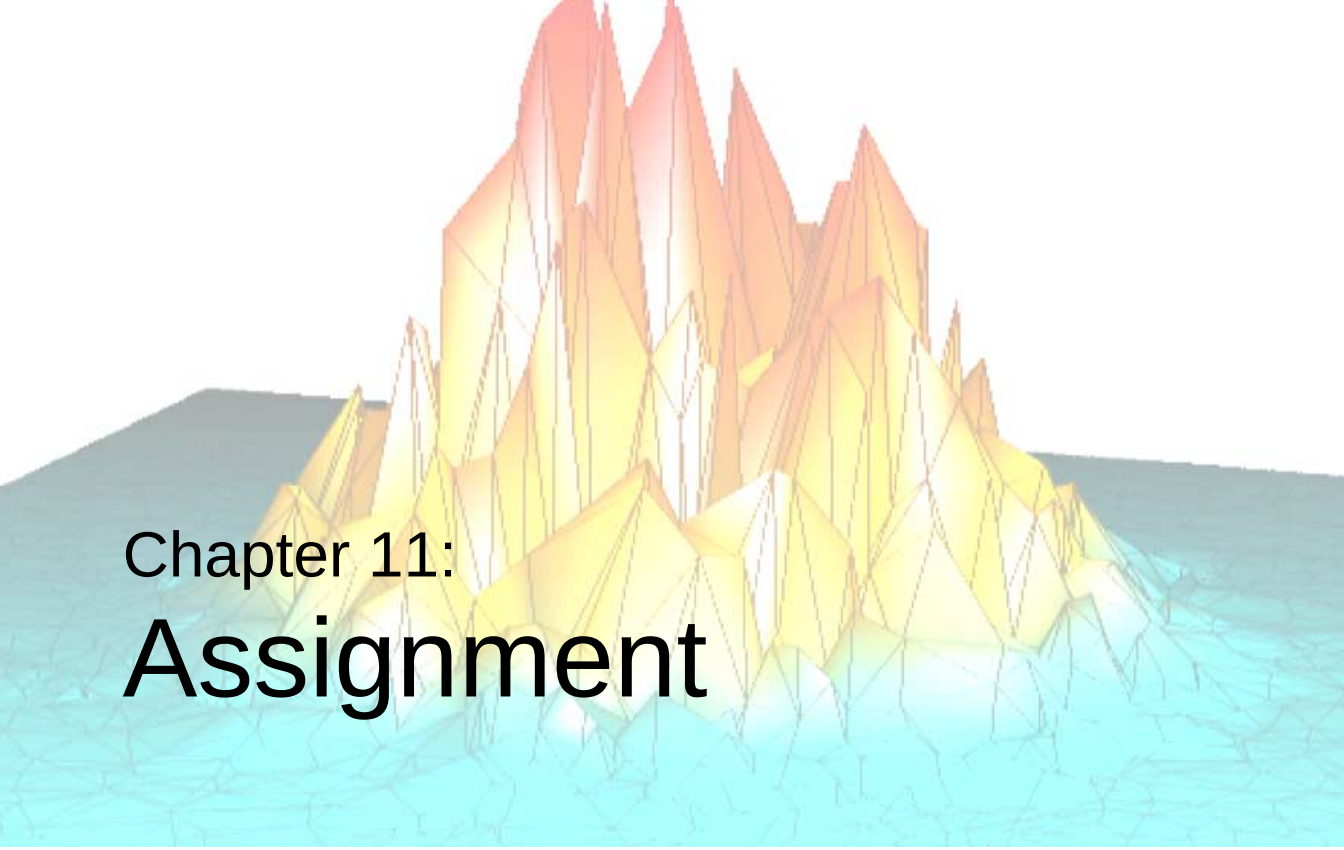
IDL supports the HDF (Hierarchical Data Format), HDF-EOS (Hierarchical Data Format-Earth Observing System), CDF (Common Data Format), and NetCDF (Network Common Data Format) self-describing, scientific data formats. Collections of built-in routines provide an interface between IDL and these formats. Documentation for specific routines and further discussion of the various formats can be found in *IDL Reference Guide*.

Support for Standard Image File Formats

IDL includes routines for reading and writing many standard graphics file formats. These routines and the types of files they support are listed in the table below. Documentation on these routines can be found in the online help (enter "?" at the IDL prompt).

Format	Read/Write Routines	Query Routine	Description
BMP	READ_BMP WRITE_BMP	QUERY_BMP	Windows Bitmap (`.bmp`) Format
Interfile	READ_INTERFILE (Write routine is n/a)	n/a	Interfile version 3.3 Format
JPEG	READ_JPEG WRITE_JPEG	QUERY_JPEG	Joint Photographic Experts Group files
NRIF	(Read routine is n/a) WRITE_NRIF	n/a	NCAR Raster Interchange Format
PICT	READ_PICT WRITE_PICT	QUERY_PICT	Macintosh version 2 PICT files (bitmap only)
PNG	READ_PNG WRITE_PNG	QUERY_PNG	Portable Network Graphics file
PPM	READ_PPM WRITE_PPM	QUERY_PPM	PPM/PGM Format
SRF	READ_SRF WRITE_SRF	QUERY_SRF	Sun Raster File
TIFF	READ_TIFF WRITE_TIFF	QUERY_TIFF	8-bit or 24-bit Tagged Image File Format
X11 Bitmap	READ_X11_BITMAP (Write routine is n/a)	n/a	X11 Bitmap format used for reading bitmaps for IDL widget button labels
XWD	READ_XWD (Write routine is n/a)	n/a	X Windows Dump format

Table 10-14: IDL-Supported Graphics Standards



Chapter 11: Assignment

The following topics are covered in this chapter:

Overview of the Assignment Statement

The assignment statement stores a value in a variable. There are three forms of the assignment statement, as shown in the following table.

Syntax	Subscript Structure	Expression Structure	Effect
Variable = *Expression*	*None*	All	*Expression* is stored in *Variable.*
Variable[*Subscripts*] = *Expression*	Scalar	Scalar	*Expression* is stored in a single element of *Variable*
	Scalar	Array	*Expression* array is inserted in *Variable* array.
	Array	Scalar	*Expression* scalar is stored in designated elements of *Variable.*
	Array	Array	Elements of *Expression* are stored in designated elements of *Variable.*

Table 11-1: Types of Assignment Statements

Syntax	Subscript Structure	Expression Structure	Effect
Variable[*Range*] = *Expression*	*Range*	Scalar	When possible, range subscripts should be avoided. See “Avoid Using Range Subscripts” on page 308. Scalar is inserted into subarray.
	Range	Array	When possible, range subscripts should be avoided. See “Avoid Using Range Subscripts” on page 308. If *Variable*[*Range*] and Array are the same size, elements of Array specified by *Range* are inserted in *Variable*. Illegal if *Variable*[*Range*] and Array are different sizes.

Table 11-1: (Continued) Types of Assignment Statements

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 128 for additional details.

Assigning a Value to a Variable

The most basic form of the assignment statement is as follows:

```
Variable = Expression
```

The old value of the variable, if any, is discarded, and the value of the expression is stored in the variable. The expression on the right side can be of any type or structure.

Examples

Some examples of the basic form of the assignment statement are as follows:

```
;Set mmax to value.
mmax = 100 * X + 2.987

;name becomes a scalar string variable.
name = 'Mary'

;Make arr a 100-element, floating-point array.
arr = FLTARR(100)

;Discard points 0 to 49 of arr. It is now a 50-element array.
arr = arr[50:*]
```

Assigning Scalars to Array Elements

The second type of assignment statement has the following form:

```
Variable[Subscripts] = Scalar_Expression
```

Here, a single element of the specified array is set to the value of the scalar expression. The expression can be of any type and is converted, if necessary, to the type of the variable. The variable on the left side must be either an array or a file variable. Some examples of assigning scalar expressions to subscripted variables are:

```
;Set element 100 of data to value.
data[100] = 1.234999

;Store string in an array. name must be a string array or an error
;will result.
name[index] = 'Joe'

;Set element [X, Y] of the 2-dimensional array image to the value
contained in pixel.
image[X, Y] = pixel
```

Using Array Subscripts

The subscripted variable can have either a scalar or array subscript. If the subscript expression is an array, the scalar value is stored in the elements of the array whose subscripts are elements of the subscript array. For example, the 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.

The WHERE function can be used to select array elements to be changed. For example, the statement:

```
data[WHERE(data LT 0)] = -1
```

sets all negative elements of data to -1 without changing the positive elements. The result of the function, WHERE(data LT 0), is a vector composed of the subscripts of the negative elements of data. Using this vector as a subscript changes only the negative elements.

Assigning Arrays to Array Elements

The third type of assignment statement has the following form:

```
Variable[Subscripts] = Array
```

Note that this form is syntactically identical to the second type of assignment statement, but that the expression on the right-hand-side is an array instead of a scalar. This form of the assignment statement is used to insert one array into another.

The array expression on the right is inserted into the array appearing on the left side of the equal sign starting at the point designated by the subscripts.

Examples

For example, to insert the contents of an array called A into array B, starting at point B[13, 24], use the following statement:

```
B[13, 24] = A
```

If A is a 5-column by 6-row array, elements B[13:17, 24:29] are replaced by the contents of array A.

In the next example, a subarray is moved from one position to another:

```
B[100, 200] = B[200:300, 300:400]
```

A subarray of B, specifically the columns 200 to 300 and rows 300 to 400, is moved to columns 100 to 200 and rows 200 to 300, respectively.

Using Array Subscripts

If the subscript expression applied to the variable is an array and an array appears on the right side of the statement:

```
Variable[Array] = Array
```

then elements from the right side are stored in the elements designated by the subscript vector. Only those elements of the subscripted variable whose subscripts appear in the subscript vector are changed. For example, the statement

```
B[[ 2, 4, 6 ]] = [4, 16, 36]
```

is equivalent to the following series of assignment statements:

```
B[2] = 4
B[4] = 16
B[6] = 36
```

Subscript elements are interpreted as if the subscripted variable is a vector. For example, if A is a $10 \times n$ matrix, the element *A[i, j]* has the subscript *i*+10**j*. The subscript array is converted to longword type before use, if necessary.

As described previously for the second form of assignment statement, elements of the subscript array that are negative or larger than the highest subscript are clipped to the target array boundaries. Note that a common error is to use a negative *scalar* subscript (e.g., A[-1]). Using this type of subscript causes an error. Negative *array* subscripts (e.g., A[[-1]]) do not cause errors.

As another example, assume that the vector DATA contains data elements and that a data drop-out is denoted by a negative value. In addition, assume that there are never two or more adjacent drop-outs. The following statements replace all drop-outs with the average of the two adjacent good points:

```
;Subscript vector of drop-outs.
bad = WHERE(data LT 0)

;Replace drop-outs with average of previous and next point.
data[bad] = (data[bad - 1] + data[bad + 1]) / 2
```

In this example, the following actions are performed:

- We use the LT (less than) operator to create an array, with the same dimensions as data, that contains a 1 for every element of data that is less than zero and a zero for every element of data that is zero or greater. We use this “drop-out array” as a parameter for the WHERE function, which generates a vector that contains the one-dimensional *subscripts* of the elements of the drop-out array that are nonzero. The resulting vector, stored in the variable bad, contains the subscripts of the elements of data that are less than zero.
- The expression data[bad - 1] is a vector that contains the subscripts of the points immediately preceding the drop-outs; while similarly, the expression data[bad + 1] is a vector containing the subscripts of the points immediately after the drop-outs.
- The average of these two vectors is stored in data[bad], the points that originally contained drop-outs.

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 subscript. The beginning and ending subscripts are separated by the colon character. An ending subscript equal to the size of the dimension minus one can be written as *.

For example, arr[I:J] denotes those points in the vector arr with subscripts between I and J inclusive. I must be less than or equal to J and greater than or equal to zero. J denotes the points in arr from arr[I] to the last point and must be less than the size of the dimension arr [I:*]. See Chapter 6, “Arrays” for more details on subscript ranges.

Examples

Assuming the variable B is a 512 × 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
```

When possible, you should avoid using range subscripts in favor of using scalar or array subscripts. Consider the following example:

```
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[4:6] = X` is much less efficient than `A[4] = X`.

Compound Assignment Operators

In addition to the standard assignment operator, IDL supports the following compound assignment operators:

##=	#=	*=	+=	-=
/=	<=	>=	AND=	EQ=
GE=	GT=	LE=	LT=	MOD=
NE=	OR=	XOR=	^=	

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
```

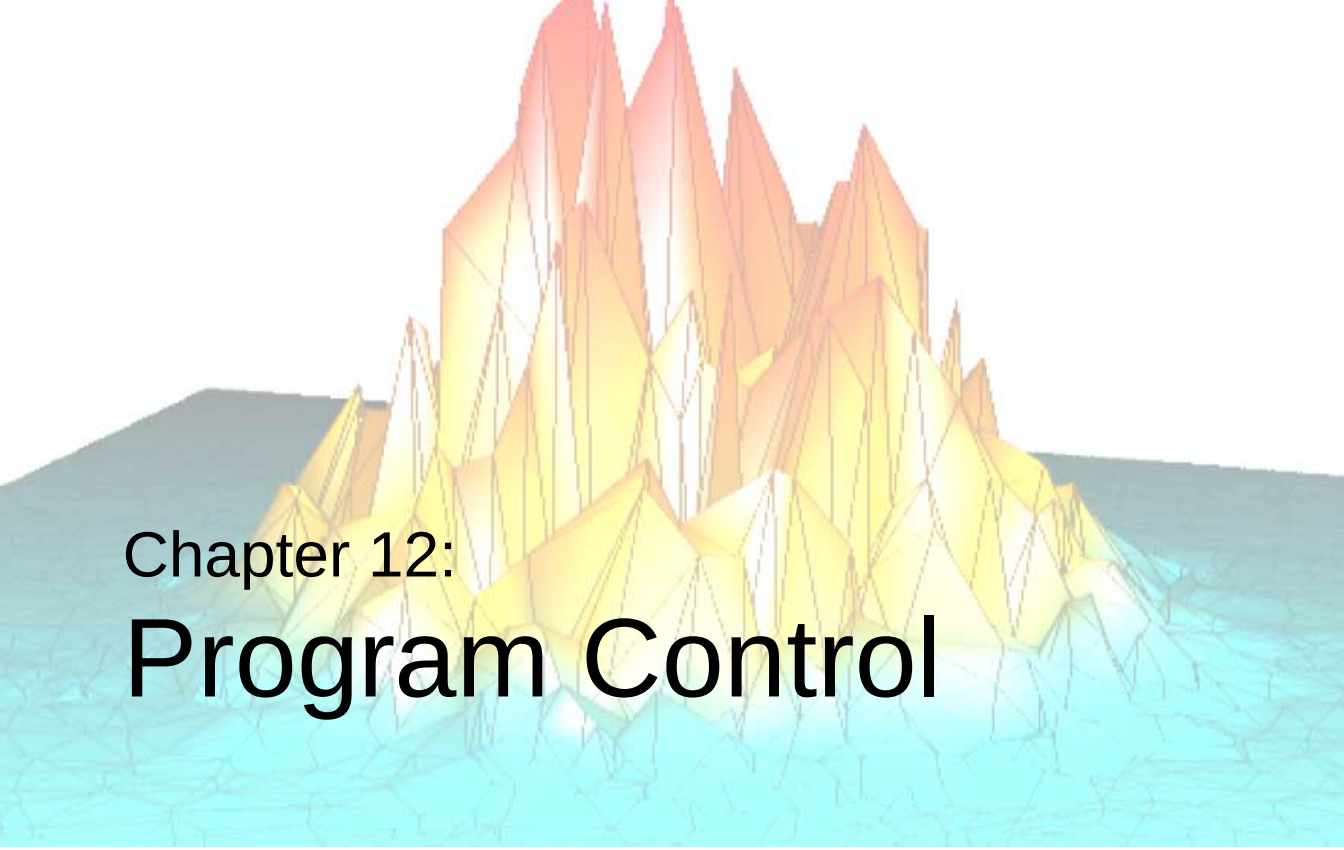
are identical, but the latter is more readable.

Using Associated File Variables

A special case occurs when using an associated file variable in an assignment statement. For additional information regarding the ASSOC function, see "ASSOC" in the *IDL Reference Guide* manual. When a file variable is referenced, the last (and possibly only) subscript denotes the record number of the array within the file. This last subscript must be a simple subscript. Other subscripts and subscript ranges, except the last, have the same meaning as when used with normal array variables.

An implicit extraction of an element or subarray in a data record can also be performed. For example:

```
;Variable A associates the file open on unit 1 with the records of
;200-element, floating-point vectors.
A = ASSOC(1, FLTARR(200))

;Then, X is set to the first 100 points of record number 2, the
;third record of the file.
X = A[0:99, 2]

;Set the 24th point of record 16 to 12.
A[23, 16] = 12

;Increment points 10 to 199 of record 12. Points 0 to 9 of the
;record remain unchanged.
A[10, 12] = A[10:*, 12]+1
```



Chapter 12: Program Control

The following topics are covered in this chapter:

Overview

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

Compound Statements

- BEGIN...END

Conditional Statements

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

Loop Statements

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

Jump 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`
		`  A=2`
		`ENDELSE`

FOR *variable=init*, *limit* DO BEGIN	ENDFOR	`FOR i=1,5 DO BEGIN`
		`  PRINT, array[i]`
		`ENDFOR`

IF *expression* THEN BEGIN	ENDIF	`IF (0) THEN BEGIN`
		`  A=1`
		`ENDIF`

REPEAT BEGIN	ENDREP	`REPEAT BEGIN`
		`  A = A * 2`
		`ENDREP UNTIL A GT B`

WHILE *expression* DO BEGIN	ENDWHILE	`WHILE ~ EOF(1) DO BEGIN`
		`  READF, 1, A, B, C`
		`ENDWHILE`

LABEL: BEGIN	END	`LABEL1: BEGIN`
		`  PRINT, A`
		`END`

case_expression: BEGIN	END	`CASE name OF`
		`'Moe': BEGIN`
		`  PRINT, 'Stooge'`
		`  END`
		`ENDCASE`

switch_expression: BEGIN	END	`SWITCH name OF`
		`'Moe': BEGIN`
		`  PRINT, 'Stooge'`
		`  END`
		`ENDSWITCH`

Table 12-1: 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" in the *IDL Reference Guide* manual for details on producing program listings with the IDL compiler.)

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

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 335 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 “Conditional Expression” on page 36.

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 315). 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

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:
`two`
	IDL Prints:
	`two`
	`three`
	`four`

Table 12-2: 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 332.) 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:

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

Achieving this behavior with CASE would be difficult.

Loop Statements

One of the most common programming tasks is to 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.

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
```

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.

FOR Statement with an Increment of One

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

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

The variable after the FOR is called the index variable and is set to the value of the first expression. The subject statement is executed, and the index variable is incremented by 1 until the index variable is larger than the second expression. This

second expression is called the limit expression. Complex limit and increment expressions are converted to floating-point type.

Warning

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

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

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

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

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

Warning

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

The following FOR statement continues infinitely:

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

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

Examples

A simple FOR statement:

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

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 & Floating-Point Operations" in Chapter 22 of the *Using IDL* manual for more information about floating-point numbers.

FOR Statement with Variable Increment

The format of the second type of FOR statement is as follows:

```
FOR Variable = Expression1, Expression2, Increment DO Statement
```

This form is used when an increment other than 1 is desired.

The first two expressions describe the range of numbers for the index variable. The Increment specifies the increment of the index variable. A negative increment allows the index variable to step downward.

Examples

The following examples demonstrate the second type of FOR statement.

```
;Decrement, K has the values 100., 99., ..., 1.
FOR K = 100.0, 1.0, -1 DO ...

;Increment by 2., loop has the values 0., 2., 4., ..., 1022.
FOR loop = 0, 1023, 2  DO ...

;Divide range from bottom to top by 4.
FOR mid = bottom, top, (top - bottom)/4.0 DO ...
```

Warning

If the value of the increment expression is zero, an infinite loop occurs. A common mistake resulting in an infinite loop is a statement similar to the following:

```
FOR X = 0, 1, .1 DO ....
```

The variable X is first defined as an integer variable because the initial value expression is an integer zero constant. Then the limit and increment expressions are converted to the type of X, integer, yielding an increment value of zero because .1 converted to integer type is 0. The correct form of the statement is:

```
FOR X = 0., 1, .1 DO ....
```

which defines X as a floating-point variable.

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 335 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

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 335 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

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:

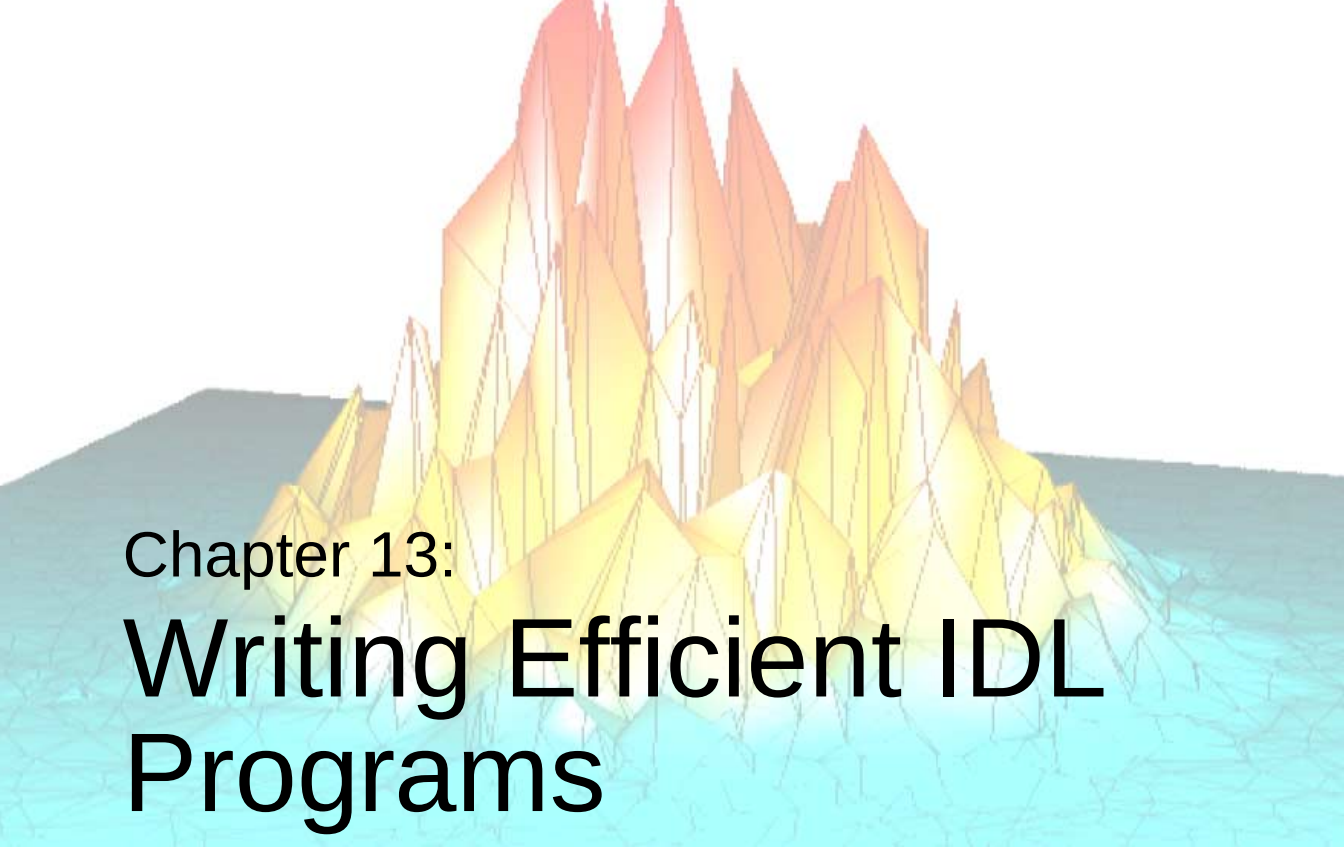
- By default:
  - Byte, integer, and long: odd integers are true, even integers are false.
  - Floating-point, and complex: non-zero values are true, zero values are false. The imaginary part of a complex number is ignored.
  - String: any string with a nonzero length is true, null strings are false.
  - Heap variables (pointers and object references): non-null values are true, null values are false.
- If the LOGICAL_PREDICATE compile option is set:
  - Numerical values: non-zero values are true, zero is false.
  - String and heap variables: non-null values are true, null values are false

See “COMPILE_OPT” in the *IDL Reference Guide* manual 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 13: Writing Efficient IDL Programs

The following topics are covered in this chapter:

Overview

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.
- Use IDL system functions and procedures wherever possible.
- Access array data in machine address order.

Attention also must be given to algorithm complexity and efficiency, as this is usually the greatest determinant of resources used.

Expression Evaluation Order

The order in which an expression is evaluated can have a significant effect on program speed. Consider the following statement, where A is an array:

```
;Scale A from 0 to 16.
B = A * 16. / MAX(A)
```

This statement first multiplies every element in A by 16 and then divides each element by the value of the maximum element. The number of operations required is twice the number of elements in A. A much faster way of computing the same result is used in the following statement:

```
;Scale A from 0 to 16 using only one array operation.
B = A * (16./MAX(A))
```

or

```
;Operators of equal priority are evaluated from left to right. Only
;one array operation is required.
B = 16./MAX(A) * A
```

The faster method only performs one operation for each element in A, plus one scalar division. To see the speed difference on your own machine, execute the following statements:

```
A = RANDOMU(seed, 512, 512)
t1 = SYSTIME(1) & B = A*16./MAX(A) & t2 = SYSTIME(1)
PRINT, 'Time for inefficient calculation: ', t2-t1
t3 = SYSTIME(1) & B = 16./MAX(A)*A & t4 = SYSTIME(1)
PRINT, 'Time for efficient calculation: ', t4-t3
```

Avoid IF Statements

Programs with array expressions run faster than programs with scalars, loops, and IF statements. Some examples of slow and fast ways to achieve the same results follow.

Example—Summing Elements

The first example adds all positive elements of array B to array A.

```
;Using a loop will be slow.
FOR I = 0, (N-1) DO IF B[I] GT 0 THEN A[I] = A[I] + B[I]

;Fast way: Mask out negative elements using array operations.
A = A + (B GT 0) * B

;Faster way: Add B > 0.
A = A + (B > 0)
```

When an IF statement appears in the middle of a loop with each element of an array in the conditional, the loop can often be eliminated by using logical array expressions.

Example—Using Array Operators and WHERE

In the example below, each element of C is set to the square-root of A if A[I] is positive; otherwise, C[I] is set to minus the square-root of the absolute value of A[I].

```
;Using an IF statement is slow.
FOR I=0,(N-1) DO IF A[I] LE 0 THEN $
   C[I]=-SQRT(-A[I]) ELSE C[I]=SQRT(A[I])

;Using an array expression is much faster.
C = ((A GT 0) * 2-1) * SQRT(ABS(A))
```

The expression (A GT 0) has the value 1 if A[I] is positive and has the value 0 if A[I]is not. (A GT 0)* 2 - 1 is equal to +1 if A[I] is positive or -1 if A[I] is negative, accomplishing the desired result without resorting to loops or IF statements.

Another method is to use the WHERE function to determine the subscripts of the negative elements of A and negate the corresponding elements of the result.

```
;Get subscripts of negative elements.
negs = WHERE(A LT 0)
;Take root of absolute value.
C = SQRT(ABS(A))
;Negate elements in C corresponding to negative elements in A.
C[negs] = -C[negs]
```

Use Vector and Array Operations

Whenever possible, vector and array data should always be processed with IDL array operations instead of scalar operations in a loop. For example, consider the problem of inverting a 512 × 512 image. This problem arises because approximately half the available image display devices consider the origin to be the lower-left corner of the screen, while the other half recognize it as the upper-left corner.

The following example is for demonstration only. The IDL system variable !ORDER should be used to control the origin of image devices. The ORDER keyword to the TV procedure serves the same purpose.

A programmer without experience in using IDL might be tempted to write the following nested loop structure to solve this problem:

```
FOR I = 0, 511 DO FOR J = 0, 255 DO BEGIN

   ;Temporarily save pixel image.
   temp = image[I, J]

   ;Exchange pixel in same column from corresponding row at bottom
   image[I, J] = image[I, 511 - J]

   image[I, 511-J] = temp

ENDFOR
```

A more efficient approach to this problem capitalizes on IDL's ability to process arrays as a single entity:

```
FOR J = 0, 255 DO BEGIN

   ;Temporarily save current row.
   temp = image[*, J]

   ;Exchange row with corresponding row at bottom.
   image[*, J] = image[*, 511-J]

   image[*, 511-J] = temp

ENDFOR
```

At the cost of using twice as much memory, processing can be simplified even further by using the following statements:

```
;Get a second array to hold inverted copy.
image2 = BYTARR(512, 512)
```

```
;Copy the rows from the bottom up.
FOR J = 0, 511 DO image2[*, J] = image[*, 511-J]
```

Even more efficient is the single line:

```
image2 = image[*, 511 - INDGEN(512)]
```

that reverses the array using subscript ranges and array-valued subscripts.

Finally, using the built-in ROTATE function is quickest of all:

```
image = ROTATE(image, 7)
```

Inverting the image is equivalent to transposing it and rotating it 270 degrees clockwise.

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.

Use Constants of the Correct Type

As explained in Chapter 3, “Constants and Variables”, 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.

Eliminate Invariant Expressions

Expressions whose values do not change inside a loop should be moved outside the loop. For example, in the loop:

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

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

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

Virtual Memory

The IDL programmer and user must be cognizant of the characteristics of virtual memory computer systems to avoid penalty. Virtual memory allows the computer to execute programs that require more memory than is actually present in the machine by keeping those portions of programs and data that are not being used on the disk. Although this process is transparent to the user, it greatly affects the efficiency of the program.

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 349. 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 × 512 byte image with a 100 kilobyte working set. The array requires 512 × 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 × 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 512×512 complex floating array requires $8*512^2$ bytes or about 2 megabytes of memory because each complex element requires 8 bytes. Deleting a variable containing a 512×512 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 × 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.

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.

The IDL Code Profiler

The IDL Code Profiler helps you analyze the performance of your applications. You can easily monitor the calling frequency and execution time for procedures and functions. The Profiler can be used with programs entered from the command line as well as programs run from within a file.

You can start the IDL Code Profiler by selecting “Profile” from the Run menu of the IDLDE or by entering PROFILER at the Command Input Line. For more information about the PROFILER procedure, see “PROFILER” in the *IDL Reference Guide* manual.

Note

Calling the Profiler from the Command Input Line does not start the Profiler dialog.

The Profile Dialog

Select “Profile” from the Run menu. The Profile dialog appears.

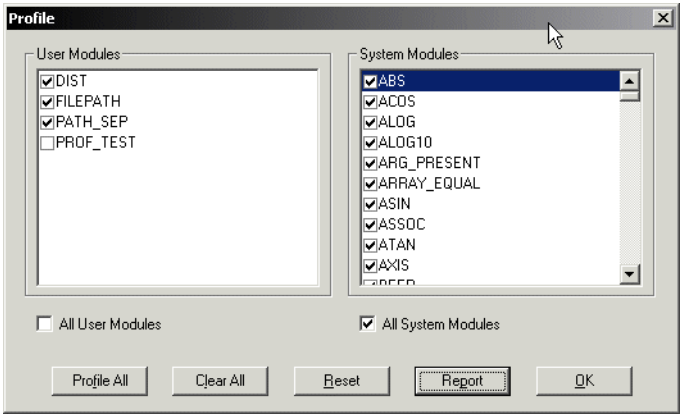


Figure 13-1: Profile Dialog

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 IDLDE. The Profile Report dialog appears.

Fields in the Profiler Report Dialog

The fields in the Profiler Report dialog show the following attributes of the modules selected for profiling from the Profile dialog. You can sort the values in each column in both ascending and descending order by clicking anywhere within the column. By default, the Modules column is sorted alphabetically.

Note

Whether you enter a program at the command line or run a program contained in a file, the PROFILER procedure 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 IDLDE. Click “Save” to save the report as a text file. The Save Profile Report dialog appears. Click “Cancel” to dismiss the Profile Report dialog. The contents remain available after cancelling. Click “Help” to display Help on this dialog.

Using the IDL Code Profiler

Open a new editor file by selecting “New” from the File menu.

Enter the following lines in the editor:

```
PRO prof_test
   OPENR, 1, FILEPATH('nyny.dat', SUBDIR=['examples', 'data'])
   a=ASSOC(1, BYTARR(768,512))
   b=a[0]
   CLOSE, 1
   TV, b
END
```

Save the file as prof_test.pro by selecting “Save” from the File menu. The Save As dialog appears.

To use the IDL Code Profiler, you must first compile the routines you would like to profile. For more involved programs, you can use RESOLVE_ALL to compile all uncompiled functions or procedures that are called in any already-compiled procedure or function.

Select “Profile...” from the Run menu. The Profile dialog appears; it will remain visible until dismissed. Select “Profile All” to profile all the available modules.

Run the application by selecting “Run” from the File menu. After the application is finished, return to the Profile dialog and click “Report”. The Profile Report dialog appears, as shown in the following figure.

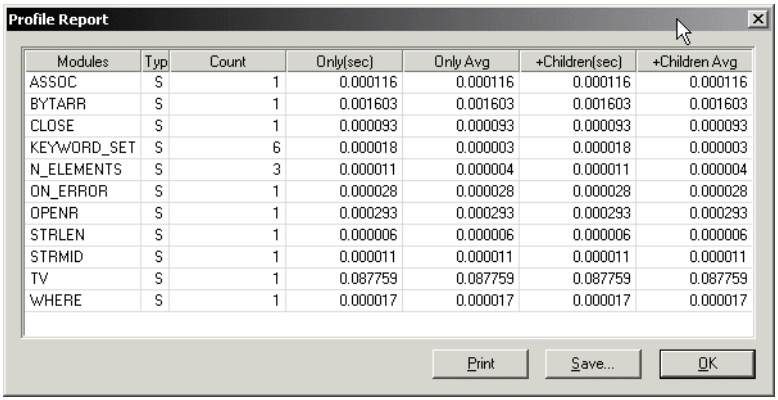


Figure 13-2: Profile Report Dialog

For more information about the capabilities of either dialog, see “The Profile Dialog” on page 352 and “The Profile Report Dialog” on page 353.

Profiling with Command Line Modules

We will demonstrate how the Profiler handles newly compiled modules. The above example set profiling for all system files, plus the user module, prof_test, and the library function, FILEPATH. If you have altered the above results, reset the report and run prof_test again.

Enter the following lines at the Command Input Line:

```
;Create a dataset using the library function DIST. Note that DIST
;is immediately compiled.
```

```
A= DIST(500)

;Display the image.
TV, A
```

Return to the Profile dialog. You will note that the DIST function has been appended to the User Module field, but that it remains 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.

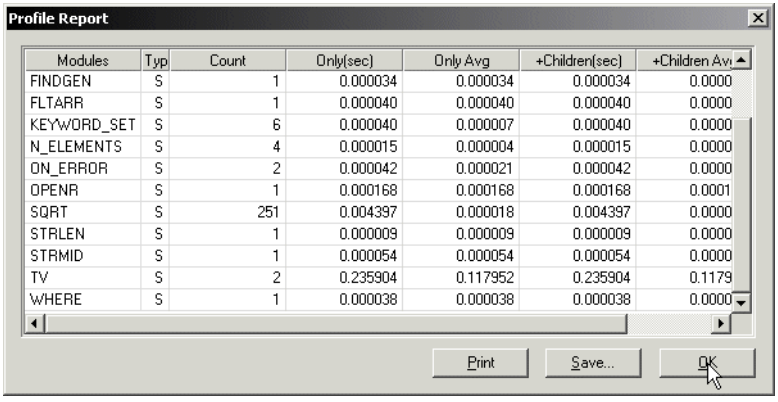
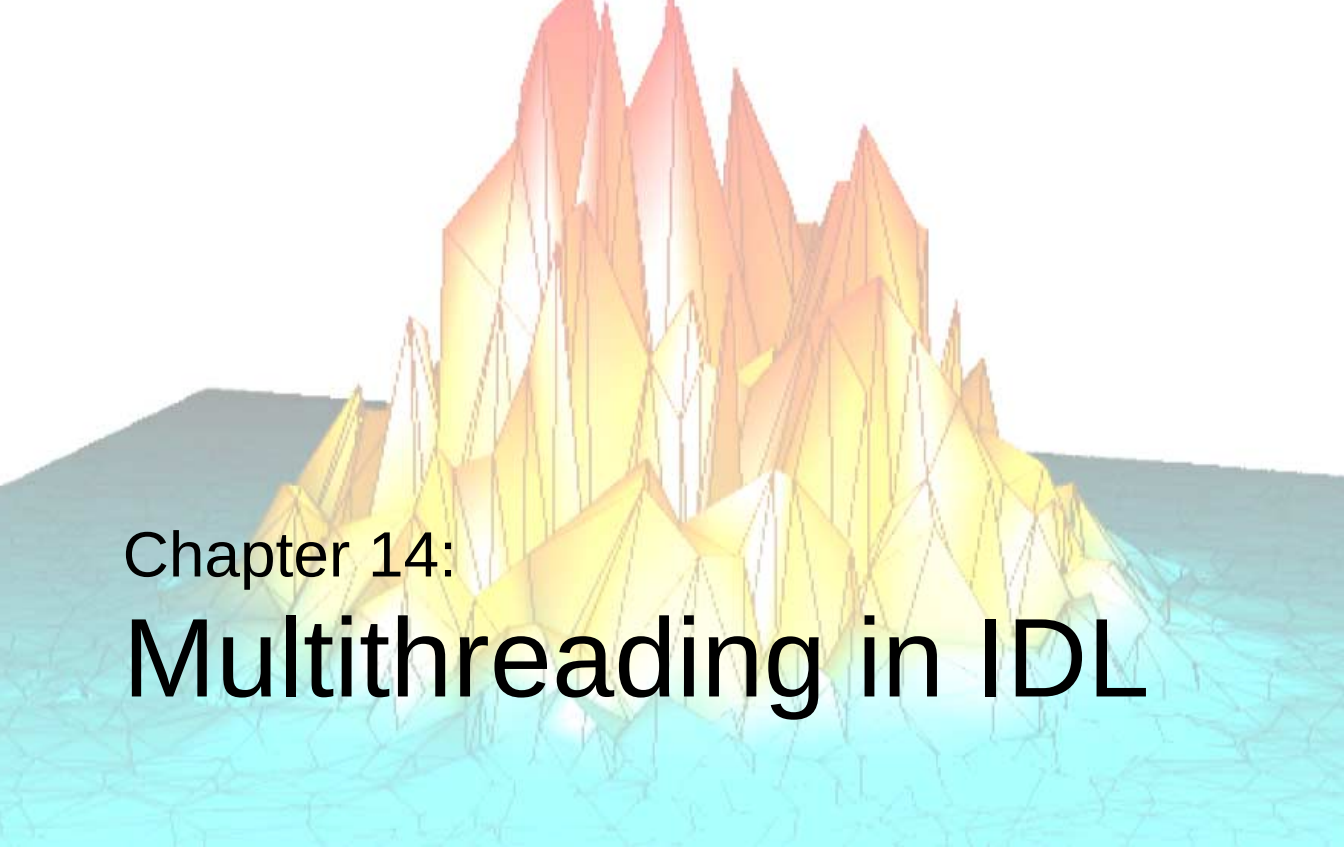


Figure 13-3: Refreshing the Profile Report

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 Input Line, you will notice that only the count for TV increases in the profiler report. You must re-enter the statement calling DIST at the Command Input Line; the already-compiled library function is executed again, making it available for profiling.



Chapter 14: 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 367.

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.

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 367.

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" in the *IDL Reference Guide* manual.

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. You can, however, 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 359 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" in the *IDL Reference Guide* manual.

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 365.

Example 2

In this example, we will:

1. Save the current thread pool settings from the !CPU system environment 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 367. 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
- ACOS
- ALOG
- ALOG10
- ASIN
- ATAN
- CEIL
- CONJ
- COS
- ERRORF
- EXP
- EXPINT
- FINITE
- FLOOR
- GAMMA
- GAUSSINT
- IMAGINARY
- ISHFT
- MATRIX_MULTIPLY
- PRODUCT
- ROUND
- SIN
- SINH
- SQRT
- TAN
- TANH
- VOIGT

- COSH
- LNGAMMA

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
- FIX
- FLOAT
- LONG
- LONG64
- UINT
- ULONG
- ULONG64

Array Manipulation Routines:

- MAX
- MIN
- REPLICATE_INPLACE
- TOTAL
- WHERE

Programming and IDL Control Routines:

- BYTEORDER
- LOGICAL_AND
- LOGICAL_OR
- LOGICAL_TRUE

Chapter 15: Solutions to Common IDL Tasks

There are various programming tasks that are often used in IDL programs. This chapter describes how to do some of the things you will commonly need to do in an IDL program. The tasks discussed in this chapter include:

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
```

Determining if a Keyword is Set

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
```

Determining the Number of Array Elements in an Expression or Variable

The N_ELEMENTS function returns the number of elements contained in any expression or variable. Scalars always have one element. The number of elements in arrays or vectors is equal to the product of the dimensions. The N_ELEMENTS function returns zero if its parameter is an undefined variable. The result is always a longword scalar.

For example, the following expression is equal to the mean of a numeric vector or array.

```
TOTAL(arr) / N_ELEMENTS(arr)
```

Determining if a Variable is Defined

The N_ELEMENTS function provides a convenient method of determining if a variable is defined. The following statement sets the variable abc to zero if it is undefined; otherwise, the variable is not changed.

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

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. 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
```

Determining the Size/Type of an Array

The determiningSIZE function returns a vector that contains information indicating the size and type of the parameter. The returned vector is always of longword type.

- The first element is equal to the number of dimensions of the parameter and is zero if the parameter is a scalar.
- The next elements contain the size of each dimension.
- After the dimension sizes, the last two elements indicate the data type and the total number of elements, respectively. The data type is encoded as follows:

Type Code	Data Type
0	Undefined
1	Byte
2	Integer (16-bit)
3	Longword integer (32-bit)
4	Floating point
5	Double-precision floating
6	Complex floating
7	String
8	Structure
9	Double-precision complex floating
10	Pointer
11	Object reference
12	Unsigned integer (16-bit)
13	Unsigned longword integer (32-bit)
14	64-bit integer
15	Unsigned 64-bit integer

Table 15-1: Type Codes Returned by the SIZE Function

The data type can also be returned by setting the TYPE keyword to SIZE. In this case, the return value of the SIZE function is the data type code of the given expression.

Examples

Example 1

Assume A is an integer array with dimensions of (3,4,5). The statements:

```
arr = INDGEN(3,4,5)
S = SIZE(arr)
```

assign to the variable S a six-element vector containing:

Element	Value	Description
S_0	3	Three dimensions
S_1	3	First dimension
S_2	4	Second dimension
S_3	5	Third dimension
S_4	2	Integer type
S_5	60	Number of elements = 3*4*5

Table 15-2: SIZE Values

The following code segment checks to see if the variable arr is two-dimensional and extracts the dimensions:

```
;Create a variable.
arr = [[1,2,3],[4,5,6]]

;Get size vector.
S = SIZE(arr)

;Check if two dimensional.
IF S[0] NE 2 THEN $
   ;Print error message.
   MESSAGE, 'Variable a is not two dimensional.'

;Get number of columns and rows.
NX = S[1] & NY = S[2]

PRINT, 'Array is ', NX, ' columns by ', NY, ' rows.'
```

IDL prints:

```
Array is    3 columns by    2 rows.
```

Example 2

The following example illustrates two ways in which to determine the type code of the input expression.

The first method requires you to access the correct element of the array returned by the SIZE function (the second to last element). For example:

```
array = [[1,2,3], [4,5,6], [7,8,9]]

sz = SIZE(array)
type = sz[3]

;A more flexible method:
sz = SIZE(array)
n = N_ELEMENTS(sz)
type = sz[n-2]
```

The second method involves using the TYPE keyword to SIZE. In this case, the value returned by the SIZE function contains only the type code of the input expression:

```
type = SIZE(array, /TYPE)
```

Determining if a Variable Contains a Scalar or Array Value

The SIZE function can also be used to determine whether a variable holds a scalar value or an array. Setting the DIMENSIONS keyword causes the SIZE function to return a 0 if the variable is a scalar, or the dimensions if the variable is an array, as shown in the following example:

```
A = 1
B = [1]
C = [1,2,3]
D = [[1,2],[3,4]]

PRINT, SIZE(A, /DIMENSIONS)
PRINT, SIZE(B, /DIMENSIONS)
PRINT, SIZE(C, /DIMENSIONS)
PRINT, SIZE(D, /DIMENSIONS)
```

IDL Prints:

```
0
1
3
2    2
```

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. Although not as flexible as the EXECUTE function (see the following page), CALL_FUNCTION and CALL_PROCEDURE 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)
```

Executing Dynamically-Created IDL Code

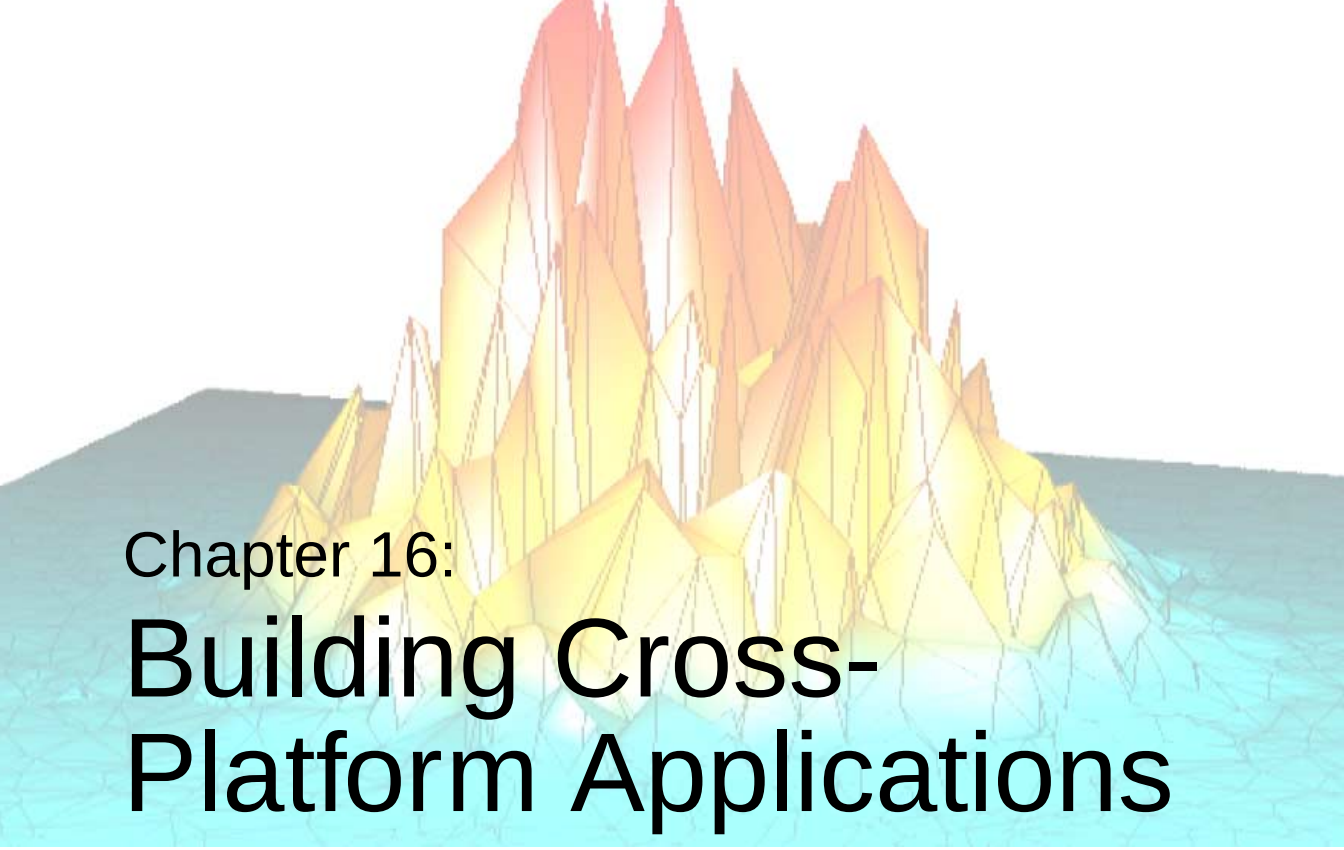
The EXECUTE function compiles and executes one or more IDL statements contained in its string parameter during runtime. EXECUTE is limited by two factors:

- Calls to EXECUTE cannot be nested, so a routine called by EXECUTE cannot use EXECUTE itself.
- The need to compile the string at runtime makes EXECUTE inefficient in terms of speed.

The CALL_FUNCTION and CALL_PROCEDURE routines provide much of the functionality of EXECUTE without imposing these limitations and should be used in preference to EXECUTE when possible.

The result of the EXECUTE function is true (1) if the string was successfully compiled and executed. If an error occurred during either phase, the result is false (0). If an error occurs, an error message is printed.

Multiple statements in the string should be separated with the “&” character. GOTO statements and labels are not allowed.



Chapter 16: Building Cross-Platform Applications

The following topics are covered in this chapter:

Overview

IDL is designed as a platform-independent environment for data analysis and programming. Because of this, the vast majority of IDL's routines operate the same way no matter what type of computer system you are using. IDL's cross-platform development environment makes it easy to develop an application on one type of system for use on any system IDL supports.

Despite IDL's cross-platform nature, there *are* differences between the computers that make up a multi-platform environment. Operating systems supply resources in different ways. While IDL attempts to abstract these differences and provide a common environment for all Windows 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.

File and Path Specifications

Different operating systems use different path specification syntax and directory separation characters. The following table summarizes the different characters used by different operating systems; see "!PATH" in the *IDL Reference Guide* manual for further details on path specification.

Operating System	Directory Separator	Path Element Separator
UNIX	/ (forward slash)	: (colon)
Windows	\ (backward slash)	; (semicolon)

Table 16-1: Directory and Path Element Separator Characters

As a result of these differences, specifying filenames and paths explicitly in your IDL application can cause problems when moving your application to a different platform. You can effectively isolate your IDL programs from platform-specific file and path specification issues by using the FILEPATH and DIALOG_PICKFILE functions.

Choosing Files at Runtime

To allow users of your application to choose a file at runtime, use the DIALOG_PICKFILE function. DIALOG_PICKFILE will always return the file path with the correct syntax for the current platform. Other methods (such as reading a file name from a text field in a widget program) may or may not provide a proper file path.

Selecting Files Programmatically

To give your application access to a file you know to be installed on the host, use the FILEPATH function. By default, FILEPATH allows you to select files that are included in the IDL distribution tree. Chances are, however, that a file you supply as part of your

own application is *not* included in the IDL tree. You can still use FILEPATH by explicitly specifying the root of the directory tree to be searched.

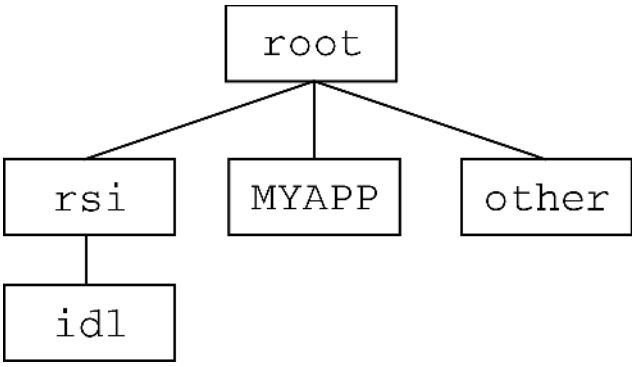


Figure 16-1: A possible directory hierarchy for an IDL application.

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:”.

Environment Variables

UNIX versions of IDL have the ability to use *environment variables* to store information about the environment in which IDL is running. Typically, environment variables are used to store information like the path to the main IDL directory, or to a batch file to be read and executed when IDL starts up. See “Environment Variables Used by IDL” in Chapter 1 of the *Using IDL* manual for details.

Microsoft Windows systems also have the ability to use environment variables to store information, but this form of information storage is much less common under Windows.

Rather than using environment variables, the IDL Development Environment stores information in *preferences*; the mechanisms used to store preferences is different between platforms, but is generally transparent to you. Configuration settings you specify in the preferences dialogs of the IDL Development Environment are saved and are available to the IDE the next time it is started.

What does this all mean in the context of writing IDL applications for multiple platforms? Simply this: don’t rely on environment variables in your programs unless you know that:

1. the target platform supports environment variables, and
2. the appropriate environment variables are defined as you wish them to be on the target platform.

Files and I/O

IDL's file input and file output routines are designed to work identically on all platforms, where possible. In the case of basic operations, such as opening a text file and reading its contents, importing an image format file into an IDL array, or writing ASCII data to a file on a hard disk, IDL's I/O routines work the same way on all platforms. In more complicated cases, however, such as reading data stored in binary data format files, different operating systems may use files that are structured differently, and extra care may be necessary to ensure that IDL reads or writes files in the proper way.

Before attempting to write a cross-platform IDL application that uses more than basic file I/O, you should read and understand the sections in Chapter 10, "Files and Input/Output" that apply to the platforms your application will support. The following are a few topics to think about when writing IDL applications that do input/output.

Byte Order Issues

Computer systems on which IDL runs support two ways of ordering the bytes that make up an arbitrary scalar: *big endian*, in which multiple byte numbers are stored in memory beginning with the most significant byte, and *little endian*, in which numbers are stored beginning with the least significant byte. The following table lists the processor types and operating systems IDL supports and their byte ordering schemes:

Processor Type	Operating System	Byte Ordering
Digital Alpha AXP	Tru64 UNIX	little-endian
Hewlett Packard PA-RISC	HP-UX	big-endian
IBM RS/6000	AIX	big-endian
Intel x86	Linux	little-endian
	Solaris x86	little-endian
	Windows	little-endian
Motorola PowerPC	Macintosh OS X and later	big-endian
SGI R4000 and up	Irix	big-endian

Table 16-2: Byte ordering schemes used by platforms that support IDL

Processor Type	Operating System	Byte Ordering
Sun SPARC	SunOS	big-endian
	Solaris	big-endian

Table 16-2: (Continued) Byte ordering schemes used by platforms that

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 272.

Logical Unit Numbers

Logical Unit Numbers (LUNs) are assigned to individual files when the files are opened by the IDL OPENR/OPENU/OPENW commands, and are used to specify which file IDL should read from or write to. There are a total of 128 LUNs available for assignment to files. While it is possible to assign any of the integers between 1-99 to a given file, when writing applications for others it is good programming practice to let IDL assign and manage the LUNs itself. By using the GET_LUN keyword to the OPEN routines, you can ask IDL to assign a free Logical Unit Number between 100-128 to the specified file. Letting IDL assign the LUN from the list of free unit numbers ensures that your application does not attempt to use a LUN already in use by someone else’s application. See the description of the GET_LUN keyword to “OPEN” in the *IDL Reference Guide* manual and “Logical Unit Numbers (LUNs)” on page 225.

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.

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 434 for details on NaN and Infinity.)

Operating System Access

While IDL provides ways to interact with each operating system under which it runs, it is not generally useful to use operating-system native functions in a cross-platform IDL program. If you find that you must use operating-system native features, be sure to determine the current operating system (as described in "Which Operating System is Running?" on page 387) and branch your code accordingly.

Display Characteristics and Palettes

Finding Screen Size

Use the GET_SCREEN_SIZE function to determine the size of the screen on which your application is displayed. Writing code that checks the screen size allows your application to handle different screen sizes gracefully.

Number of Colors Available

Use the N_COLORS and TABLE_SIZE fields of the !D system variable to determine the number of colors supported by the display and the number of color-table entries available, respectively.

Make sure that your application handles relatively small numbers of colors (less than 256, say) gracefully. For example, Microsoft Windows reserves the first 20 colors out of all the available colors for its own use. These colors are the ones used for title bars, window frames, window backgrounds, scroll bars, etc. If your application is running on a Windows machine with a 256-color display, it will have at most 236 colors available to work with.

Similarly, make sure that your application handles TrueColor (24-bit or 32-bit color) displays as well. If your application uses IDL's color tables, for example, you will need to force the application into 8-bit mode using the command

```
DEVICE, DECOMPOSED=0
```

to use indexed-color mode on a machine with a TrueColor display.

Fonts

IDL uses three font systems for writing characters on the graphics device, whether that device be a display monitor or a printer: Hershey (vector) fonts, TrueType (outline) fonts, and device (hardware) fonts. Fonts are discussed in detail in Appendix H, “Fonts” in the *IDL Reference Guide* manual.

Both TrueType and Vector fonts are displayed identically on all of the platforms that support IDL. This means that if your cross-platform application uses either the TrueType fonts supplied with IDL or the Vector fonts, there is no need for platform-dependent code.

Printing

IDL displays operating-system native dialogs using the DIALOG_PRINTJOB and DIALOG_PRINTERSETUP functions. Since the dialogs that control printing and printer setup differ between systems, so do the options and capabilities presented via IDL's print dialogs. If your IDL application uses IDL's printing dialogs, make sure that your interface calls the dialog your user will expect for the platform in question.

SAVE and RESTORE

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 also that if you are restoring a file created with VAX IDL version 1, you must restore on a machine running VMS.

Widgets

IDL's user interface toolkit is designed to provide a "native" look and feel to widget-based IDL applications. Where possible, widget toolkit elements are built around the operating system's native dialogs and controls; as a result, there are instances where the toolkit behaves differently from operating system to operating system. This section describes a number of platform-dependencies in the IDL widget toolkit. Consult the descriptions of the individual DIALOG and WIDGET routines in the *IDL Reference Guide* for complete details.

Dialog Routines

IDL's DIALOG_ routines (DIALOG_PICKFILE, etc.) rely on operating system native dialogs for most of their functionality. This means, for example, that when you use DIALOG_PICKFILE in an IDL application, 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" in the *IDL Reference Guide* manual 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 387. 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” in the *IDL Reference Guide* manual 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” in the *IDL Reference Guide* manual 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.

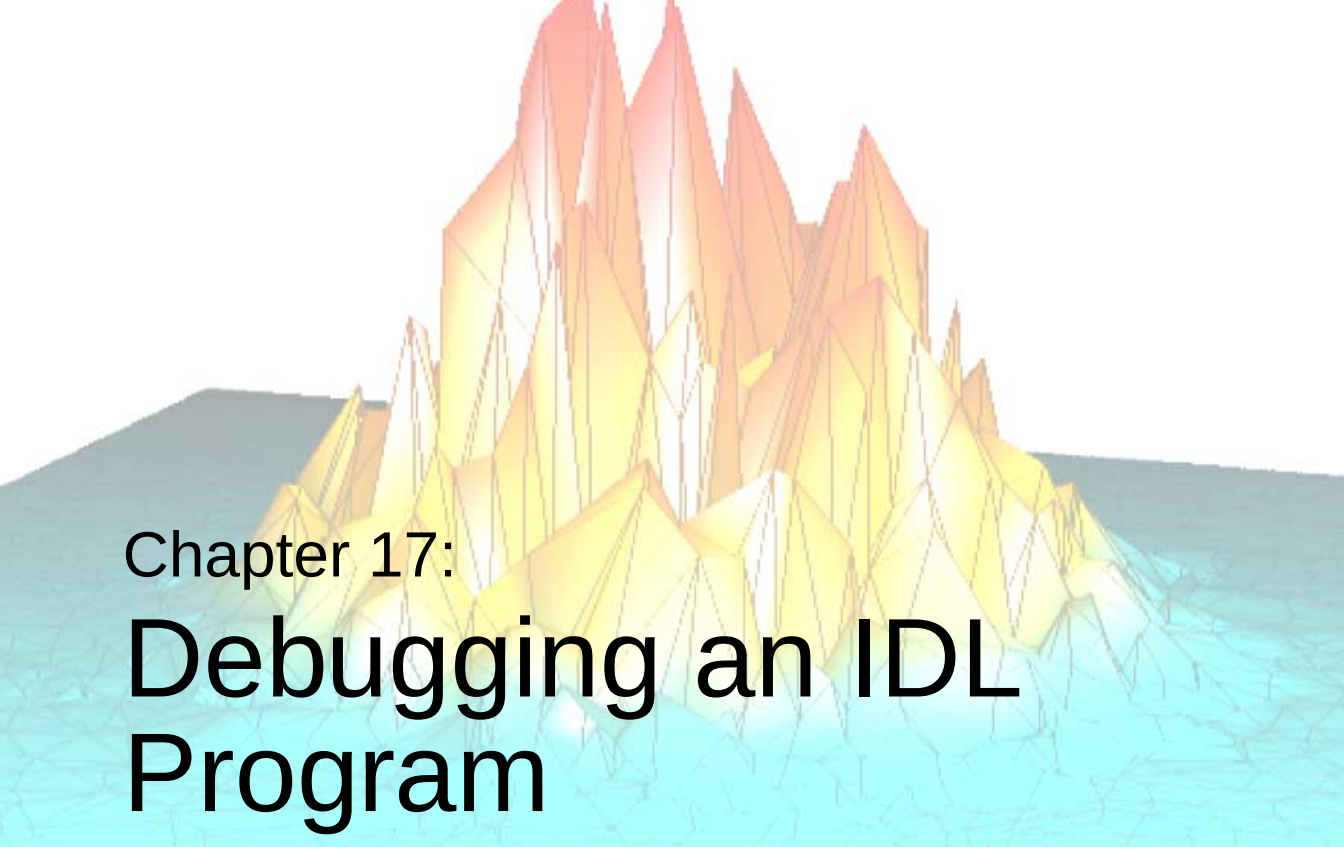
Handle individual widget events carefully, and be sure to test your code on all platforms supported by your application.

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 17: Debugging an IDL Program

The following topics are covered in this chapter:

Overview

There are several tools you can use to help you find errors in your IDL code. The Run menu item in the IDL Development Environment provides several ways to access IDL's built-in debugging and executive commands. The Variable Watch Window helps you keep track of the variables used in your program.

This chapter explains the debugging commands and contains short examples using the IDLDE interface to debug a file.

Debugging Commands

When a file displayed in an IDL editor window has been compiled (by selecting **Compile** or **Memory Compile** from the **Run** menu, or by entering `.COMPILE`, `.COMPILE -f`, or `.RUN` at the IDL command prompt), a number of debugging commands become available for selection. For more information on the Run menu, see "Run Menu" in Chapter 2 of the *Using IDL* manual.

When execution is interrupted, a current-line indicator is placed next to the line that will be executed when processing resumes. The routine being compiled need not already be shown in an editor window. If a routine compiled with the `.RUN`, `.RNEW`, or `.COMPILE` executive commands contains an error, IDLDE will display the file automatically.

A Simple Example

A simple procedure, called BROKEN, has been included in the IDL distribution. An error occurs when BROKEN is executed.

Start the IDLDE. Call the BROKEN procedure by entering:

```
BROKEN
```

at the IDL command line. An error is reported in the Output Log window and an editor window containing the file BROKEN.PRO appears.

A "Variable is undefined" error has occurred. Since execution stopped at line 4, that line is highlighted with an arrow.

There are several ways of fixing this error. We could edit the program file to explicitly define the variable i, or we could change the program so that it accepts a parameter at the command line. We can also define the variable i on the fly and continue execution of the program without making any changes to the program file. We'll do this first, then go back and edit the program to accept a command-line parameter.

To define the variable `i` and assign it the value 10, click in the IDL command line and enter:

```
i = 10
```

Step Through the Program

Select **Step Into** from the **Run** menu to execute line 4 with the new value of `i` and step to the next program line.

The Output Log reports:

```
10
```

The current-line pointer advances to the next line in the window containing the file BROKEN.PRO. You could continue stepping through the program by choosing **Step Into** repeatedly (or by entering `.STEP` at the IDL command prompt).

The **Trace Execution** dialog offers an opportunity to automatically step through the program. Select **Trace...** from the **Run** menu. The **Trace Execution** dialog appears.

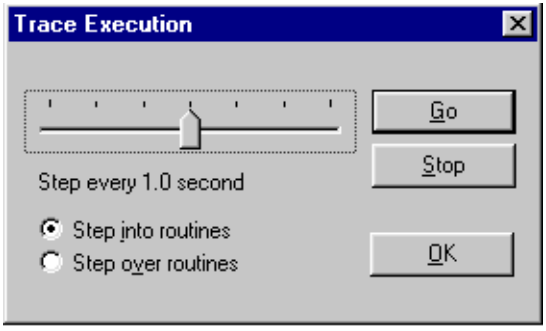


Figure 17-1: Trace Execution Dialog (Windows)

Figure 17-2: Trace Execution Dialog (Unix)

Click **Go** or **Run** to automatically issue the `.STEP` command until the `END` statement is encountered, or click **Stop** to halt trace execution. Moving the slider in the **Trace Execution** dialog controls the length of the pauses between step commands. You can also select whether to step into routines, executing successive .STEP commands at each line (Windows only), or to step over routines, issuing successive .STEPOVER

commands. For more information, see ".STEP" and ".STEPOVER" in the *IDL Reference Guide* manual. Click **OK** or **Dismiss** to dismiss the dialog.

You can also continue execution of the program without stepping through. Select **Run** from the **Run** menu, noting that the Output Log shows that IDL calls broken. Define the variable i in the Command Input Line. Select **Run** again. The Output Log now shows that IDL calls .CONTINUE. IDL prints the resulting output to the Output Log window:

```
10
20
30
40
```

When stepping through a main program, if the next line calls another IDL procedure or function, you have three options with which to handle execution of the nested program. Selecting **Step Into** executes statements in order by successive Step commands. Selecting **Step Over** executes statements to the end of the called function, without interactive capability. Select **Step Out** to continue processing until the main program returns.

Fix the Program

To fix the program permanently, edit the first line of the program to read:

```
PRO BROKEN, i
```

Select **Save** from the **File** menu and **Compile** from the **Run** menu. IDL saves the modified text file over the old version and compiles the modified routine. To call this new version of BROKEN with an input argument of 10, enter:

```
BROKEN, 10
```

The Output Log window prints the result:

```
10
20
30
40
```

Breakpoints

You can suspend execution of a program temporarily by setting breakpoints in the code. Set a breakpoint at the fifth line of BROKEN.PRO by placing the cursor in the line that reads:

```
PRINT, i*2
```

and selecting **Set Breakpoint** from the **Run** menu. A breakpoint dot appears next to the line. Now enter:

```
BROKEN, 10
```

The Output Log window displays the following:

```
10
% Breakpoint at: BROKEN              5
```

and a current line indicator arrow marks line 5. Select **Run** to resume execution. To list the breakpoints, enter `HELP,/BREAKPOINT` at the command line.

Setting a breakpoint allows you to inspect (or change) variable definitions as the program executes. Since our example does not set any variables, setting a breakpoint in BROKEN.PRO is not very informative. Breakpoints can be extremely helpful, though, when debugging complex programs, or programs that call other routines. For more information on working with breakpoints, see the following section, "Working with Breakpoints".

Working with Breakpoints

You can select to edit, enable/disable, and change breakpoint properties using Breakpoint Toolbar buttons. Additionally, through the Edit Breakpoints dialog, breakpoints can be set for execution dependent upon a condition or enabled after the breakpoint has been encountered a specific number of times.

The Breakpoint Toolbar Buttons

There are three buttons in the main menu bar. These are:

The **Toggle Breakpoint** button creates or deletes a breakpoint. Create a breakpoint at the line where your cursor is positioned by clicking the Toggle Breakpoint button. If a breakpoint already exists in the line where your cursor is positioned, clicking this button removes the breakpoint.

The **Enable/Disable Breakpoint** button enables or disables a breakpoint. If a breakpoint is enabled, a filled circle appears next to the line in the IDL Editor window. If disabled, the circle is not filled. Disabled breakpoints are ignored when you run the file.

The **Edit Breakpoints** button displays the Edit Breakpoints dialog. In previous releases, this printed a listing of the current breakpoints. From this dialog, you can list your current breakpoints, create new breakpoints, enable or disable breakpoints, change breakpoint options, or delete breakpoints.

The Windows Edit Breakpoints Dialog

The **Edit Breakpoints** dialog allows you to add, remove, and remove all breakpoints in a file as well as the ability to move to the line in the source file that contains the breakpoint. The following figure shows the **Edit Breakpoints** dialog:

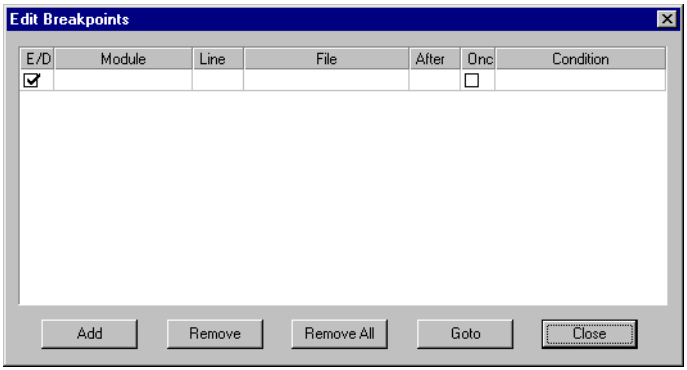


Figure 17-3: Edit Breakpoints Dialog

To create a breakpoint using the **Edit Breakpoints** dialog, complete the following steps:

1. Open the file you in which you want to set a breakpoint.
2. Display the **Edit Breakpoints** dialog by clicking the  button in the IDLDE Tool Bar or by selecting **Run** → **Edit Breakpoints...**
3. Place the cursor in the line in which you want to create the breakpoint in the Editor window.
4. Select **Add** in the **Edit Breakpoints** dialog box. You will see a new entry display in the dialog. The following table describes each property of a breakpoint:

Item	Description
E/D	Specifies whether a breakpoint is enabled or disabled. If a check mark is displayed, the breakpoint is enabled and execution will stop when the all criteria for the breakpoint is met.
Module	Specifies the procedure or function where the breakpoint is set. **Note -** This item will not be displayed until the file has been compiled with the new breakpoint.
Line	Specifies the line number where the breakpoint occurs.
File	Specifies the filename where the breakpoint occurs.
After	Specifies how many times the execution must pass the breakpoint before stopping execution. For example, if this item is set to 0, execution will stop the first time this breakpoint is encountered. If it is set to 9, execution will not stop until the breakpoint has been encountered for the ninth time.
Once	The breakpoint is removed after it is encountered for the first time.
Condition	Specifies a condition to be met for the execution to stop. The condition is a string containing an IDL expression. When a breakpoint is encountered, the expression is evaluated. If the expression is true (if it returns a non-zero value), program execution is interrupted. The expression is evaluated in the context of the program containing the breakpoint.

Table 17-1: Edit Breakpoints Description

5. At this point, you can modify any of the items (except Module and Line) by double-clicking in the entry.

Your breakpoint entry is now complete. When you run your program, execution is halted at the breakpoints you have specified.

The Variable Watch Window

The Variable Watch window displays current variable values after IDL has completed execution. If the calling context changes during execution — as when stepping into a procedure or function — the variable table is replaced with a table appropriate to the new context. While IDL is at the main program level, the Watch window remains active and displays any variables created.

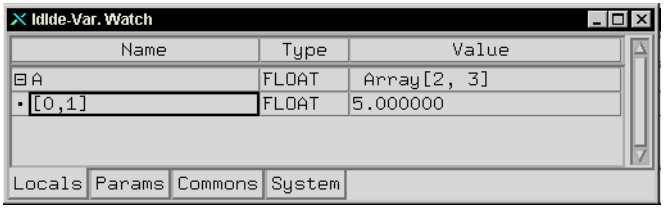


Figure 17-4: Variable Watch Window

Customizing Variable Watch Window Layout

To hide the Variable Watch window, select **Window** → **Hide Variable Watch**. Select **Show Variable Watch** to make it reappear. Changing the Window menu will only affect the current IDL session.

To apply your changes to future sessions, select **File** → **Preferences** and click the **Layout** tab. In the section labeled Show Windows, select or clear check boxes associated with the windows you want to appear. Click **Apply** to save your changes for future IDL sessions and **OK** to exit.

Note

Selection or clearing of **Window** menu items reflects changes in the **Layout** preferences and vice versa.

The Variable Watch Interface Description

The Variable Watch window is refreshed after the IDLDE has completed execution. Each Variable Watch window contains the following folders:

- Locals

This tab contains descriptions of local variables. Local variables are created from IDL's main program level. For example, entering a=1 at the Command Input Line lists the integer a in the Locals tab.

- Params

  This tab contains descriptions of parameters. The variables and expressions passed to a function or procedure are parameters. For more information, see "Parameters" on page 74.

- Commons

  This tab contains descriptions of variables contained in common blocks. The name of each common block is shown in parentheses next to the variable contained within it. For more information, see "Common Blocks" on page 63.

- System

  This tab contains descriptions of system variables. System variables are a special class of predefined variables available to all program units. For more information about system variables, see Appendix D, "System Variables" in the *IDL Reference Guide* manual.

Each tab contains a table listing the attributes of the variables included in the category. You can size the columns by clicking on the line to the right of the title of the column you wish to expand or shrink. Drag the mouse either left or right until you are satisfied with the width of the column. For example, to change the width of the Name column, click and drag on the line separating the Name field from the Type field.

The following fields describe variable attributes:

- Name

  This field shows the name of the variable. This field is read-only, except for array subscript descriptions (see example in Using the Variable Watch Window below).

  For compound variables such as arrays, structures, pointers, and objects, click the "+" symbol to the left of the name to show the variables included in the compound variable. Click the "-" symbol to collapse the description.

- Type

  This field shows the type of the variable. This field is read-only.

- Value

This field shows the value of the variable. To edit a value in UNIX, highlight the cell by clicking on it, press the function key F2 to enter editing mode, and enter the new value. To edit a value in Windows, double click on the cell to highlight it and enter the new value.

The Name, Type, and Value fields are displayed as when using the HELP procedure. For more information about variables, see "Variables" on page 59.

The Variable Watch Window and Objects

Object references are expanded only if they reference non-null objects. Object data are expanded only if the object method has finished running. Object data are read-only and cannot be changed with the Variable Watch window.

Using the Variable Watch Window

Arrays are expanded to show one array element. Click on the "+" symbol next the name of the array to display the initial array subscript. You can change this field to display the characteristics of any other array element.

Note

To enter editing mode in Motif, press F2 after clicking on the cell to be edited. In Windows, double click on the cell.

To edit the subscript, highlight the cell by clicking on it, and modify the name using the arrow keys to maneuver. For example, enter the following:

```
;Create an array with 2 columns and 3 rows.
A=MAKE_ARRAY(2,3)

;Show the values of array A in the Output Log. They will all be
;zero.
PRINT, A

;Assign the value of 5 to the value in the array subscripted as 2.
;This is the same as entering A(0,1)=5.
A(2)=5

;Show the new values of array A.
PRINT, A
```

IDL prints:

```
0.00000      0.00000
5.00000      0.00000
0.00000      0.00000
```

It is easy to manipulate variables within the Watch window. Click on the "+" expansion symbol next to the array A. The subscript [0,0] will be revealed beneath the description of A. Enter editing mode and change [0,0] to [0,1].

Press **Enter** to effect the change. Notice that the value of the subscript is displayed as 5, as you entered from the command line. Press the **Tab** key to highlight the value of the subscript [0,1]. You can change it to another number. Enter [1,0] in the subscript name field. You can also change the value from 0.00000 to another number.

For more information about arrays, see Chapter 6, "Arrays".

Chapter 18: Controlling Errors

The following topics are covered in this chapter:

Overview

This chapter discusses routines and methods used to check and handle errors that occur in IDL programs. The routines covered here are rarely used interactively.

IDL divides possible execution errors into three categories: input/output, math, and all others. There are three main error-handling routines: CATCH, ON_ERROR, and ON_IOERROR. CATCH is a generalized mechanism for handling exceptions and errors. The ON_ERROR routine handles regular errors when an error handler established by the CATCH procedure is not present. The ON_IOERROR routine allows you to change the default way in which input/output errors are handled. The FINITE and CHECK_MATH routines provide control over math errors.

Default Error-Handling Mechanism

In the default case, whenever an error is detected by IDL during the execution of a program, program execution stops and an error message is printed. The execution context is that of the program unit (procedure, function, or main program) in which the error occurred.

Sometimes it is possible to recover from an error by manually entering statements to correct the problem. Possibilities include setting the values of variables, closing files, etc., and then entering the command .CONTINUE, which resumes execution of the program unit at the beginning of the statement that caused the error.

As an example, if an error occurs because an undefined variable is referenced, you can simply define the variable from the keyboard, then continue execution with .CON. Of course, this is a temporary solution. You should still edit the program file to fix the problem permanently.

Disappearing Variables

IDL users may find that all their variables have seemingly disappeared after an error occurs inside a procedure or function. The misunderstood subtlety is that after the error occurs, IDL's context is *inside the called procedure*, not in the main level. All variables in procedures and functions, with the exception of parameters and common variables, are local in scope. Typing RETURN or RETALL will make the lost variables reappear.

RETALL is best suited for use when an error is detected in a procedure and it is desired to return immediately to the main program level despite nested procedure calls. RETALL issues RETURN commands until the main program level is reached.

The HELP command can be used to see the current call stack (i.e., which program unit IDL is in and which program unit called it). For more information, see "HELP" in the *IDL Reference Guide* manual.

Controlling Errors Using CATCH

The CATCH procedure provides a generalized mechanism for handling any type of errors and exceptions within IDL. Calling CATCH establishes an error handler for the current procedure that intercepts all errors that can be handled by IDL, with the exception of non-fatal warnings such as math errors (e.g., floating-point underflow). The CATCH mechanism is similar to C's setjmp/longjmp facilities or C++'s catch/throw facilities.

When an error occurs, each active procedure, beginning with the offending procedure and proceeding up the call stack to the main program level, is examined for an error handler (established by a call to CATCH). If an error handler is found, control resumes at the statement after the call to CATCH. The index of the error is returned in the argument to CATCH and is also stored in !ERROR_STATE.CODE. The associated error message is stored in !ERROR_STATE.MSG. If no error handlers are found, program execution stops, an error message is issued, and control reverts to the interactive mode.

For more information, see "CATCH" and "!ERROR_STATE" in the *IDL Reference Guide* manual.

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.

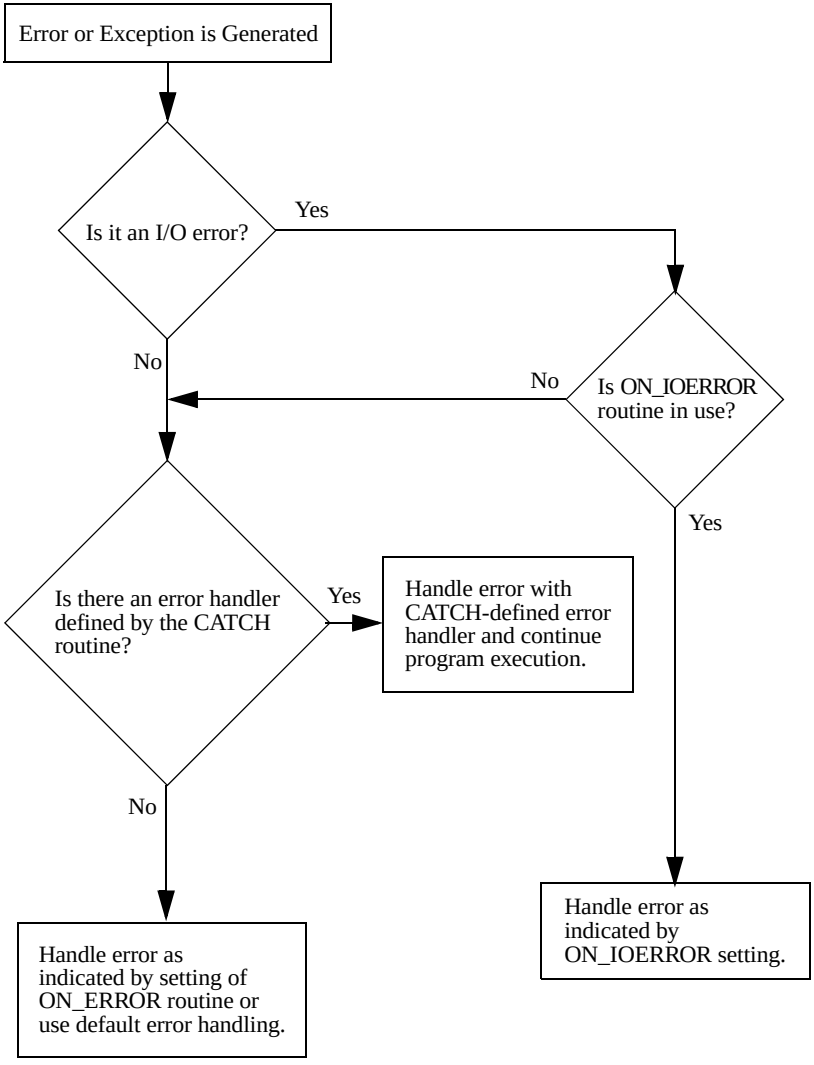


Figure 18-1: Error Handling in IDL.

Canceling an Error Handler

Call CATCH with the CANCEL keyword set to cancel a procedure's error handler. This cancellation does not effect other error handlers that may be established in other active procedures.

Generating an Exception

To generate an exception and cause control to return to the error handler, use the MESSAGE procedure. Calling MESSAGE generates an exception that sets the !ERROR_STATE system variable. !ERROR_STATE.MSG is set to the string used as an argument to MESSAGE. See "Error Signaling" on page 428.

Example Using CATCH

The following procedure illustrates the use of CATCH:

```
PRO ABC

;Define variable A.
A = FLTARR(10)

;Establish error handler. When errors occur, the index of the error
;is returned in the variable Error_status. Initially, this
;argument is set to zero.
CATCH, Error_status

;This statement begins the error handler.
IF Error_status NE 0 THEN BEGIN

   PRINT, 'Error index: ', Error_status
   PRINT, 'Error message:', !ERR_STRING

   ;Handle the error by extending A.
   A=FLTARR(12)
   CATCH, /CANCEL
ENDIF

A[11]=12     ;Cause an error.

;Even though an error occurs in the line above, program execution
;continues to this point because the event handler extended the
;definition of A so that the statement can be re-executed.
HELP, A

END
```

Running the ABC procedure causes IDL to produce the following output and control returns to the interactive prompt:

```
Error index:         -101
Error message:
Attempt to subscript A with <INT (      11)> is out of range.
A               FLOAT     = Array(12)
```

Controlling Errors Using ON_ERROR

The ON_ERROR procedure determines the action taken when an error is detected inside a user procedure or function and no error handlers established with the CATCH procedure are found. The possible options for error recovery are shown in the following table:

Value	Action
0	Stop immediately in the context of the procedure or function that caused the error. This is the default action.
1	Return to the main program level and stop.
2	Return to the caller of the program unit that called ON_ERROR and stop.
3	Return to the program unit that called ON_ERROR and stop.

Table 18-1: Error Recovery Options

One useful option is to use ON_ERROR to cause control to be returned to the caller of a procedure in the event of an error. The statement:

```
ON_ERROR, 2
```

placed at the beginning of a procedure will have this effect. Include this statement in library procedures and other routines that will be used by others once the routines have been debugged. This form of error recovery makes debugging a routine difficult because the routine is exited as soon as an error occurs; therefore, it should be added once the code is completely tested.

Note that error handlers established by CATCH supersede calls to ON_ERROR made in the same procedure.

Controlling Input/Output Errors

The default action for handling input/output errors is to treat them exactly like regular errors and follow the error handling strategy set by ON_ERROR. You can alter this default by using the ON_IOERROR procedure to specify the label of a statement to which execution should jump if an input/output error occurs. When IDL detects an input/output error and an error-handling statement has been established, control passes directly to the given statement without stopping program execution. In this case, no error messages are printed.

Note that calls to ON_IOERROR made in the procedure that causes an I/O error supersede any error handling mechanisms created with CATCH and the program branches to the label specified by ON_IOERROR.

When writing procedures and functions that are to be used by others, it is good practice to anticipate and handle errors caused by the user. For example, the following procedure segment, which opens a file specified by the user, handles the case of a nonexistent file or read error.

```
;Define a function to read, and return a 100-element,
;floating-point array.
FUNCTION READ_DATA, FILE_NAME

;Declare error label.
ON_IOERROR, BAD

;Use the GET_LUN keyword to allocate a logical file unit.
OPENR, UNIT, FILE NAME, /GET_LUN

A = FLTARR(100)     ;Define data array.

READU, UNIT, A      ;Read the data array.

;Clean up and return.
GOTO, DONE

;Exception label. Print the error message.
BAD:  PRINT, !ERR_STRING

;Close and free the input/output unit.
DONE: FREE_LUN, UNIT

;Return the result. This will be undefined if an error occurred.
RETURN, A

END
```

The important things to note in this example are that the FREE_LUN procedure is always called, even in the event of an error, and that this procedure always returns to its caller. It returns an undefined value if an error occurs, causing its caller to encounter the error.

Error Signaling

The MESSAGE procedure is used by user procedures and functions to issue errors. It has the form:

```
MESSAGE, Text
```

where *Text* is a scalar string that contains the text of the error message.

The MESSAGE procedure issues error and informational messages using the same mechanism employed by built-in IDL routines. By default, the message is issued as an error, the message is output, and IDL takes the action specified by the ON_ERROR procedure.

As a side effect of issuing the error, appropriate fields of the system variable !ERROR_STATE are set; the text of the error message is placed in !ERROR_STATE.MSG, or in !ERROR_STATE.SYS_MSG for the operating system's component of the error message. See "Error Handling" on page 431 or "!ERROR_STATE" in the *IDL Reference Guide* manual for more information.

As an example, assume the statement:

```
MESSAGE, 'Unexpected value encountered.'
```

is executed in a procedure named CALC. IDL would print:

```
% CALC: Unexpected value encountered.
```

and execution would halt.

The MESSAGE procedure accepts several keywords that modify its behavior. See "MESSAGE" in the *IDL Reference Guide* manual 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. Everything displayed, except for the word "test," is part of the IDL message definition.

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" in the *IDL Reference Guide* manual for additional details. Specify the BLOCK and NAME keywords to the MESSAGE procedure to issue a message from a message block you have defined.

Obtaining Traceback Information

It is sometimes useful for a procedure or function to obtain information about its caller(s). The HELP procedure returns, in a string array, the contents of the procedure stack when the CALLS keyword parameter is specified. The first element of the resulting array contains the module name, source filename, and line number of the current level. The second element contains the same information for the caller of the current level, and so on, back to the level of the main program.

For example, the following code fragment prints the name of its caller, followed by the source filename and line number of the call:

```
HELP, CALLS = A

;print 2nd element
PRINT, 'called from:', A[1]
```

This results in a message of the following form:

```
Called from: DIST </usr2/idl/lib/dist.pro (27)>
```

Programs can readily parse the traceback information to extract the source file name and line number.

Error Handling

IDL contains a system variable that is updated when errors occur. This system variable is described below.

!ERROR_STATE

This system variable is a structure. Whenever an error occurs, IDL sets the fields in this system variable according to the nature of the field. An IDL error is always comprised of an IDL-generated component, and may also contain an operating system-generated component.

The fields for the !ERROR_STATE system variable are described below:

- NAME — A read-only string variable containing the error name of the IDL-generated component of the last error message. Although the error code—as defined below in CODE—may change between IDL sessions, the name will always remain the same. If an error has not occurred in the current IDL session, this field is set to IDL_M_SUCCESS.
- BLOCK — A read-only string variable containing the name of the message block for the IDL-generated component of the last error message. If an error has not occurred in the current IDL session, this field is set to IDL_MBLK_CORE.
- CODE — The error code of the IDL-generated component of the last error in IDL. Whenever an error occurs, IDL sets this system variable to the error code (a negative integer number) of the error. Although the error code may change between IDL sessions, the name—as defined above in NAME—will always remain the same. If an error has not occurred in the current IDL session, this field is set to 0.
- SYS_CODE — The error code of the operating system-generated component, if it exists, of the last error. IDL sets this system variable to the OS-defined error code.

  For historical reasons, SYS_CODE is a two-element longword array. The first element of the array (that is, SYS_CODE[0]) contains the OS-defined error code. The second element of the array is not used, and always contains zero. Either !ERROR_STATE.SYS_CODE or !ERROR_STATE.SYS_CODE[0] will return the relevant error code.

- SYS_CODE_TYPE: A string describing the type of system code contained in SYS_CODE. A null string in this field indicates that there is no system code corresponding to the current error. The possible non-NULL values are:

Value	Meaning
errno	Unix/Posix system error
win32	Microsoft Windows Win32 system error
winsock	Microsoft Windows sockets library error

Table 18-2: System error code types.

- MSG — The error message of the IDL-generated component of the last error. Whenever an error occurs, IDL sets this field to the error message (a scalar string) that corresponds to the error code. If an error has not occurred in the current IDL session, this field is set to the null string, ' '.
- SYS_MSG — The error message of the operating system-generated component, if it exists of the last error. When an operating system error occurs, IDL sets this field to the OS-defined error message string. If an error has not occurred in the current IDL session, this field is set to the null string, ' '.
- MSG_PREFIX — A string variable containing the prefix string used for the IDL-generated component of error messages.

Using !ERROR_STATE

At the beginning of an IDL session, !ERROR_STATE contains default information. To see this information, you can either view !ERROR_STATE from the System field of the Variable Watch Window (see "The Variable Watch Window" on page 413) or you can enter PRINT, !ERROR_STATE at the Command Input Line. After an error has occurred, all of the fields of !ERROR_STATE display their updated status.

You can use MESSAGE, /RESET_ERROR STATE to reset all the fields in !ERROR_STATE to their default values.

Math Errors

The detection of math errors, such as division by zero, overflow, and attempting to take the logarithm of a negative number, is hardware and operating system dependent. Some systems trap more errors than other systems. On systems that implement the IEEE floating-point standard, IDL substitutes the special floating-point values NaN and Infinity when it detects a floating point math error. (See "Special Floating-Point Values" on page 434.) Integer overflow and underflow is not detected. Integer divide by zero is detected on all platforms.

A Note on Floating-Point Underflow Errors

Floating-point underflow errors occur when a non-zero result is so close to zero that it cannot be expressed as a normalized floating-point number. In the vast majority of cases, floating-point underflow errors are harmless and can be ignored. For more information on floating-point numbers, see "Accuracy & Floating-Point Operations" in Chapter 22 of the *Using IDL* manual.

Accumulated Math Error Status

IDL handles math errors by keeping an accumulated math error status. This status, which is implemented as a longword, contains a bit for each type of math error that is detected by the hardware. When IDL automatically checks and clears this indicator depends on the value of the system variable !EXCEPT. The CHECK_MATH function also allows you to check and clear the accumulated math error status when desired.

!EXCEPT has three possible values:

!EXCEPT=0

Do not report exceptions.

!EXCEPT=1

The default. Report exceptions when the IDL interpreter returns to an interactive prompt. Any math errors that occurred since the last interactive prompt (or call to CHECK_MATH) are printed in the IDL command log. A typical message looks like:

```
% Program caused arithmetic error: Floating divide by 0
```

!EXCEPT=2

Report exceptions after each IDL statement is executed. This setting also allows IDL to report on the program context in which the error occurred, along with the line number in the procedure. A typical message looks like:

```
% Program caused arithmetic error: Floating divide by 0
% Detected at  JUNK                3 junk.pro
```

Special Floating-Point Values

Machines which implement the IEEE standard for binary floating-point arithmetic have two special values for undefined results: NaN (Not A Number) and Infinity. Infinity results when a result is larger than the largest representation. NaN is the result of an undefined computation such as zero divided by zero, taking the square-root of a negative number, or the logarithm of a non-positive number. In many cases, when IDL encounters the value NaN in a data set, it treats it as “missing data.” The special values NaN and Infinity are also accessible in the read-only system variable !VALUES. These special operands propagate throughout the evaluation process—the result of any term involving these operands is one of these two special values. For example:

```
;Multiply NaN by 3
PRINT, 3 * !VALUES.F_NAN
```

IDL prints:

```
NaN
```

It is important to remember that the value NaN is literally not a number, and as such cannot be compared with a number. For example, suppose you have an array that contains the value NaN:

```
A = [1.0, 2.0, !VALUES.F_NAN]
PRINT, A
```

IDL prints:

```
1.00000      2.00000          NaN
```

If you try to select elements of this array by comparing them with a number (using the WHERE function, for example), IDL will generate an error:

```
;Print the indices of the elements of A with a value greater than
;one.
PRINT, WHERE(A GT 1.0)
```

IDL prints:

```
1
```

```
% Program caused arithmetic error: Floating illegal operand
```

To avoid this problem, use the FINITE function to make sure arguments to be compared are in fact valid floating-point numbers:

```
PRINT, WHERE(FINITE(A) EQ 1)
```

IDL prints the indices of the finite elements of A:

```
0           1
```

To then print the indices of the elements of A that are both finite and greater than 1.0, you could use the command:

```
PRINT, WHERE(A[WHERE(FINITE(A) EQ 1)] GT 1.0)
```

IDL prints:

```
1
```

Similarly, if you wanted to find out which elements of an array were *not* valid floating-point numbers, you could use a command like:

```
;Print the indices of the elements of A that are not valid
;floating-point numbers.
PRINT, WHERE(FINITE(A) EQ 0)
```

IDL prints:

```
2
```

Note that the special value Infinity *can* be compared to a floating point number. Thus, if:

```
B = [1.0, 2.0, !VALUES.F_INFINITY]
PRINT, B
```

IDL prints:

```
1.00000      2.00000          Inf
```

and

```
PRINT, WHERE(B GT 1.0)
```

IDL prints:

```
1           2
```

You can also compare numbers directly with the special value Infinity:

```
PRINT, WHERE(B EQ !VALUES.F_INFINITY)
```

IDL prints:

```
2
```

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 669.

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 19: Providing Online Help For Your Application

The following topics are covered in this chapter:

Overview

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 19-1: 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.

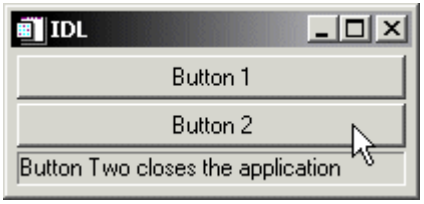


Figure 19-2: A status line.

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 444.

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” in the *IDL Reference Guide* manual 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” on page 787 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 IDL's Online Help Viewers" on page 449.

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" in the *IDL Reference Guide* manual.

About IDL's Online Help System

IDL uses different online help systems on UNIX and Windows platforms; the differences are described in detail below. All versions of IDL include the complete IDL documentation set in Adobe Portable Document Format (PDF).

The Full IDL Documentation Set in PDF

The complete IDL documentation set is available in a set of Adobe Portable Document Format (PDF) files. Adobe Systems Inc. created the Portable Document Format in the early 1990s, basing it on their PostScript language. PDF is intended to allow documents to be displayed in exactly the same manner on a wide variety of computing platforms.

The IDL PDF files are electronic representations of the individual books in the documentation set, and can be either viewed on screen or printed (in full or in part) on a local printer. When viewed on-screen, the PDF books provide hyperlinked cross-references, tables of contents, and indices, allowing for speedy navigation through the set. In addition, some versions of the Adobe Acrobat software provide a fast full-text search capability, using a pre-compiled full-text index of the entire document set.

Viewing PDF files requires a separate application, not included in the IDL installation. Various versions of the Adobe Acrobat application are in wide use, and may already be installed on your system. The Adobe Acrobat Reader application is available free of charge from Adobe at `http://www.adobe.com`. In addition, the most current version of Acrobat Reader available for each platform supported by IDL at the time of IDL's release is included on the IDL CD-ROM.

Note

Other third-party PDF viewers (notably GhostScript) are available, but UNIX versions of IDL *require* that a version of Acrobat Reader be installed in order for integration with the `?` command and ONLINE_HELP procedure to work. See "UNIX Online Help" on page 447 for details.

The PDF version of the documentation set is available on all platforms, in the `Help` subdirectory of the IDL distribution.

Microsoft Windows Help

On Microsoft Windows systems, IDL uses the Windows native HTML Help system to display context-sensitive help and most documentation for IDL language features

and routines. In addition, the HTML Help files in the IDL online help system contain hypertext links to the full IDL documentation set in PDF format.

Note

In order to display the PDF version of the documentation set, the Adobe Acrobat plug-in for Microsoft Internet Explorer must be installed on the system. The plug-in, along with the full Acrobat Reader application, is available at no charge directly from Adobe at `http://www.adobe.com`.

Note

Viewing PDF files within the HTML Help viewer (or any other Web browser Acrobat plug-in) disables the full-text-search mechanism in the Adobe Acrobat software. If you wish to use this feature of the PDF documentation set, you must open the IDL documentation files in a stand-alone version of Acrobat, such as the free Acrobat Reader. See "Portable Document Format Files" on page 451 for information on launching a stand-alone copy of Acrobat from within IDL.

UNIX Online Help

On UNIX systems, IDL's online help system launches a copy of the Adobe Acrobat software to display help information. As a result, using IDL's online help system on a UNIX system requires that a version of Adobe Acrobat be installed on the system, and that the corresponding `acroread` command be available in a directory included in your UNIX `PATH` environment variable. See the *Installing and Licensing IDL* manual for details on installing and configuring Acrobat for use with IDL.

In addition, on most UNIX platforms, IDL uses an Adobe Acrobat *plug-in* that handles communication between IDL and Acrobat. The plug-in allows you to control some Acrobat functions from within IDL, most notably specifying a search term along with the IDL `?` command — see "IDL's Acrobat Plug-In" below for details.

On platforms where the IDL Acrobat plug-in is not available, the `?` command (with or without a search term) will cause IDL to launch Acrobat and open the PDF documentation set to a default page. You can then use the navigation facilities of the PDF documents and Acrobat itself to locate the appropriate topic.

IDL's Acrobat Plug-In

The IDL Acrobat plug-in is a piece of software provided by RSI that handles communication between IDL and the version of Acrobat installed on the system. On platforms that support the plug-in, it is installed along with IDL and launched

automatically when you use either the ? command or the ONLINE_HELP procedure. The plug-in provides the following features:

- It allows you to specify a search topic to the ? command or ONLINE_HELP procedure. If the specified topic exists, it will be displayed in Acrobat. If the topic does not exist, a default topic is displayed instead.

  For example, specifying

  ```
  ? FFT
  ```

  at the IDL command line on a UNIX system that supports the plug-in will cause Acrobat to display the entry for the FFT function from the *IDL Reference Guide*.

- It allows IDL to remain in communication with a single instance of Acrobat, changing the displayed page based on input from IDL (via either the ? command or the ONLINE_HELP procedure).

- It allows IDL to close the Acrobat application, either via the QUIT keyword to the ONLINE_HELP procedure or when IDL itself exits.

For a variety of reasons, some IDL platforms do not support the IDL Acrobat plug-in. On these platforms, IDL will launch a copy of Acrobat if one is available, optionally passing the name of a PDF file for Acrobat to open. IDL has no avenue of communication with the Acrobat process after it has been launched, however, and none of the above-listed features are available. For example, specifying

```
? FFT
```

at the IDL command line on a UNIX system that does not support the plug-in will cause IDL to display an error message and Acrobat to display the first page of the *IDL Reference Guide*.

Note

On UNIX platforms, Acrobat is launched by either the ? command or the ONLINE_HELP procedure. Under Microsoft Windows, Acrobat is only launched if the ONLINE_HELP procedure is called with the name of a PDF file as the value of the BOOK keyword.

As of IDL 5.6, the IDL Acrobat plug-in is ***not*** available for the following platforms:

- AIX
- Mac OS X
- Microsoft Windows

Using IDL's Online Help Viewers

You can use IDL's online help system to display help files of your own creation. 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.

- Microsoft Windows Help
- Portable Document Format Files
- HTML Files

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" in the *IDL Reference Guide* manual 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.

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 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 format, and caution your Windows users that they must have a PDF-viewing application installed in order to use your help file.
- Create a single help file in HTML format, and caution all of your users that they must have a Web browser installed to use your help system. In addition,

UNIX users must ensure that the browser is properly configured for use by IDL, as described in "Displaying HTML Files under UNIX" under "ONLINE_HELP" in the *IDL Reference Guide* manual.

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 that while UNIX versions of IDL require some version of Adobe Acrobat Reader, Windows versions of IDL will launch whatever application is associated with PDF files when ONLINE_HELP is called with a PDF file as the value of the BOOK keyword.

Note

On all platforms, 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.

Note on PDF Named Destinations

In a Windows Help system, a specific topic within a help file can be displayed by specifying its *context number* in the call to ONLINE_HELP. In a PDF file, *named destinations* play a similar role. If you have created named destinations in your PDF file, IDL can take advantage of them — *provided the IDL Acrobat plug-in is installed*. (See "IDL's Acrobat Plug-In" on page 447 for additional information on the plug-in.) Depending on the status of the plug-in and the named destination specified, IDL will behave in one of the following ways:

- If the plug-in is installed and a valid named destination is supplied via the *Value* argument to ONLINE_HELP, the specified topic will be displayed immediately.
- If the plug-in is installed and an invalid named destination is supplied via the *Value* argument to ONLINE_HELP, the first page of the file will be displayed.
- If the plug-in is not installed, the first page of the file will be displayed regardless of the *Value* argument.

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, and optionally specify a named destination in the *Value* argument. 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" in the *IDL Reference Guide* manual 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 named destination "controls":

```
ONLINE_HELP, 'controls', BOOK='path\myapp.pdf', /FULL_PATH
```

where *path* is the full path to the file `myapp.pdf`. Note that on platforms that do not support the IDL Acrobat plug-in, the PDF viewer will display the first page of `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.

Cross-Platform Issues

If you intend to use PDF files to supply online help for your cross-platform application, keep the following things in mind:

- Windows installations of IDL do not require that Acrobat or any other PDF-viewing application be installed. Warn your users that they will need a PDF viewer to use your online help.
- Not all IDL platforms support the IDL Acrobat plug-in. On platforms that do not support the plug-in, IDL will not be able to automatically display named destinations specified as the *Value* argument to ONLINE_HELP. In addition, you will not be able to close the PDF viewer launched by IDL programmatically from within IDL.

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" in the *IDL Reference Guide* manual 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" in the *IDL Reference Guide* manual 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:

```
ONLINE_HELP, 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`.

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 Files under UNIX" under "ONLINE_HELP" in the *IDL Reference Guide* manual 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.

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: `.chm` (Windows only), `.hlp` (Windows only), `.pdf`, `.html`, `.htm`. You can override this behavior by explicitly specifying the desired file extension.

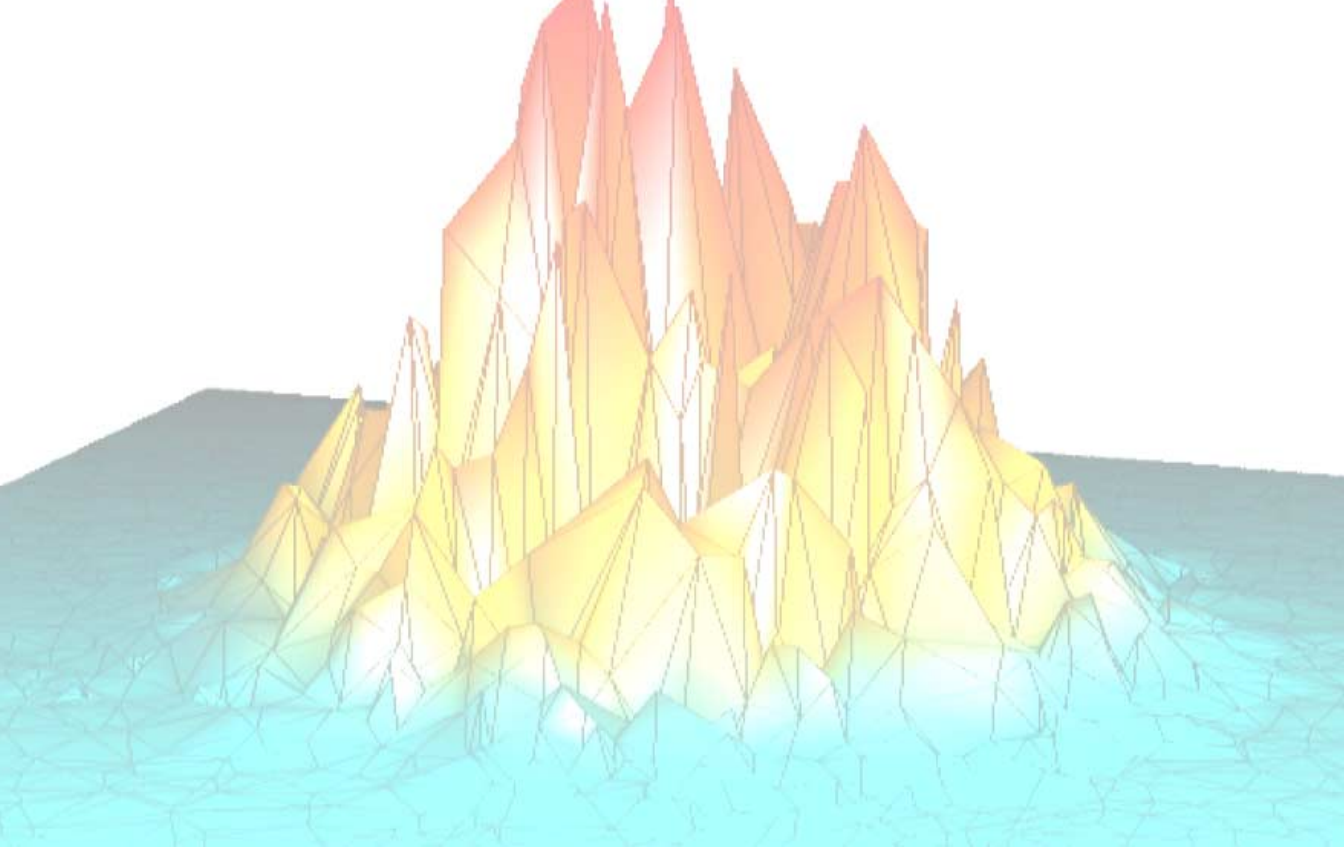
By default, !HELP_PATH contains the `help` subdirectory of the main IDL directory. To change the default value of !HELP_PATH, set the IDL_HELP_PATH environment variable using one of the following procedures:

- Under UNIX, set the IDL_HELP_PATH environment variable using a normal shell command.
- Under Windows, set the IDL_HELP_PATH environment variable using the System control panel.

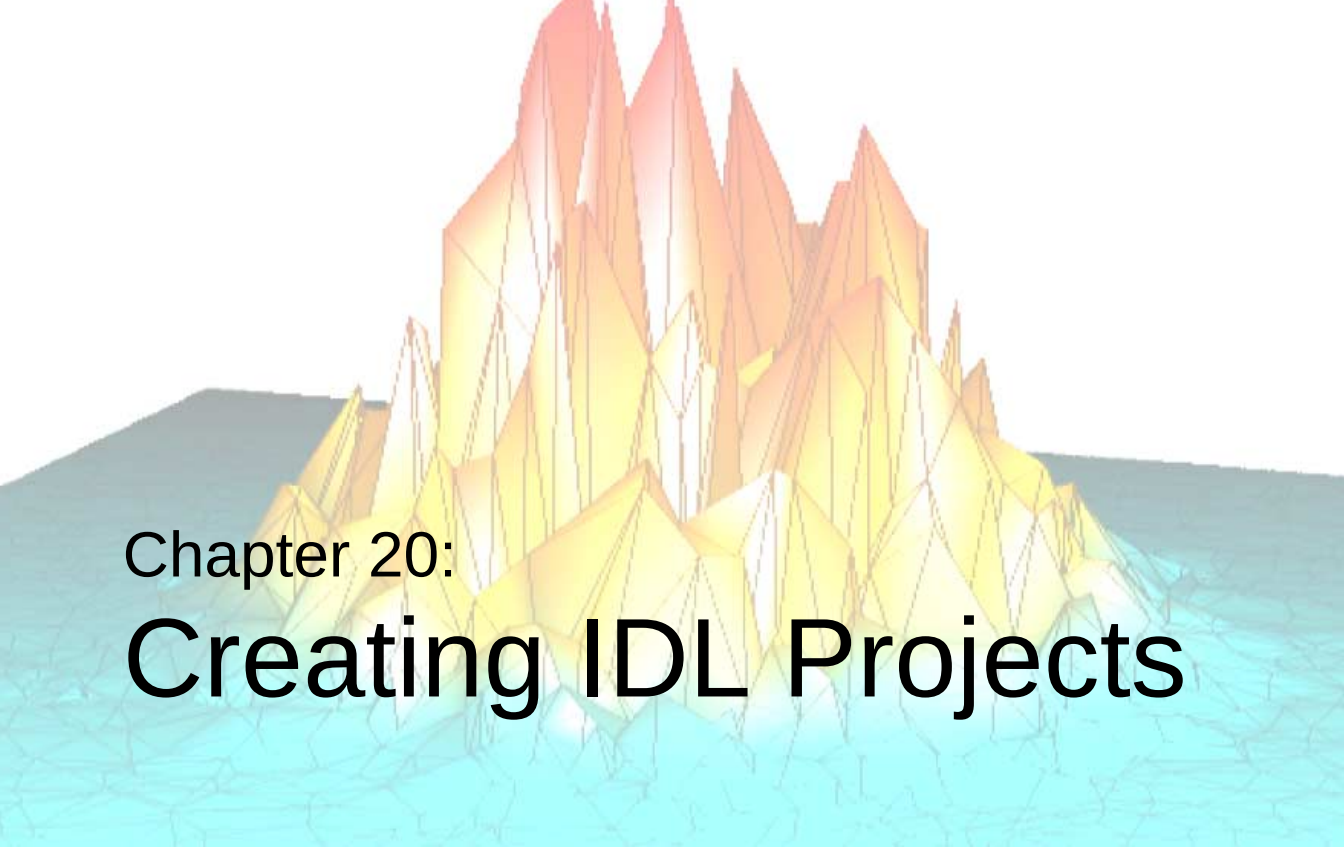
To change the value of !HELP_PATH during a single IDL session, simply assign a new value to the system variable. For example, to add a directory of your choice to the end of the default Help path, you could use the following command:

```
!HELP_PATH=!HELP_PATH+'mypath'
```

where *mypath* is a valid path string, including the appropriate path element separator character for your platform.



Part III: Creating Applications in IDL



Chapter 20: Creating IDL Projects

This chapter describes the following topics.

Overview

IDL Project allows you to easily develop applications in IDL for distribution among other developers, colleagues, or users who have IDL. If you want to develop applications for users who do not have IDL previously installed on their computer, contact your Research System sales representatives for more information on how you can distribute an IDL Runtime version.

Working with an IDL Project allows you to easily prepare your IDL application for distribution among other developers, colleagues, or users. You can organize, manage, compile, run, and create distributions of all of your application files from within the IDL Project interface. An IDL Project simplifies the process of preparing your application for distribution by offering a visual interface to application files and by automatically creating the script necessary for distributing a Runtime version of IDL. Whether you have existing files that you want to package as an application or you are building an application from the ground up, IDL Project offers the flexibility and functionality you need in a development environment.

Access to all Files in Your Application

An IDL Project has an easy to use visual interface that allows clear organization to all of the required files you need for your IDL application. This includes source files, data files, image files, or any other files your application will need to run. By default, an IDL Project contains the following categories for your files:

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

You can also create your own folders or rename existing folders to customize your IDL Project.

Working with an IDL Project

An IDL Project makes it easy to add, remove, move, edit, compile, and run your application. Additionally, Project saves all of your workspace information including breakpoints set in source code. Since breakpoints are saved when you save your project, this alleviates the need to reset them every time you open a source code file in

your project. If you save and exit your project with open files, those same files will be automatically opened when you re-open the project.

You can easily access files in your project by simply double-clicking on them. Source (`.pro`) files are opened in the IDL Editor and `.prc` (IDL GUIBuilder) files are opened in the IDL GUIBuilder. By holding down CTRL and left-clicking or by holding down SHIFT and left-clicking, you can select multiple files in the IDL Project window. You can then edit, move, compile, delete, or set the properties of multiple files at one time.

Compiling and Running Your Application

Compiling and running applications is fast and easy. Through the Project menu, you can compile all of your source files or just the files that you have modified before running your application.

Build Your Application

This feature allows you to quickly test your application. Building your application creates an IDL `.sav` file that contains all of the programs in your application. If you have `.prc` (IDL GUIBuilder) files in your project, they will also be compiled and the generated source (.pro) and the event (`*_eventcb.pro`) files will be automatically added to your IDL Project.

Exporting Your Applications

Once you have completed your application, you can quickly and easily create a distribution for your application so that you can distribute it to colleagues or customers. There are options for exporting either compiled code or source code. All your source code or compiled code (`.sav` files), IDL GUIBuilder files, data files, and image files are copied to a directory you specify.

You can also create an IDL Runtime distribution to include with your application. If you are interested in sharing your application with users who do not have IDL, please contact your RSI sales representative to discuss the options available to you.

The IDL Project Interface

The IDL Project window displays the contents of your current project and allows you to manipulate your project.

Note

If you are not using your IDL Project, you can hide the IDL Project window by selecting **File → Preferences** and then clicking the Layout tab. Under the Show Windows section, deselect the Project checkbox. When you open or create an IDL Project, the Project window will automatically be displayed and this preference will be reset to selected.

If you click the plus sign to expand your project, you will see the groups in your project. If you click the plus sign on a folder, you will see the individual files that are grouped in that folder.

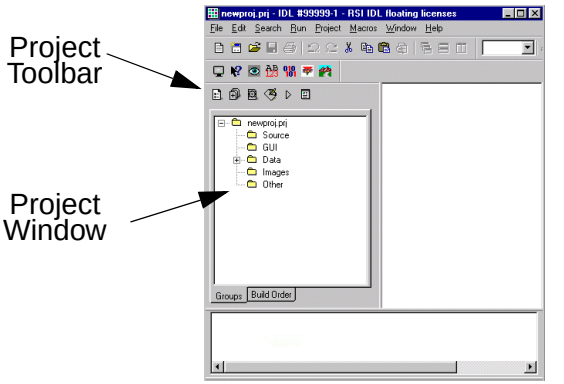


Figure 20-1: The Project Window

If you have added a file to a project and then either removed or renamed it on your system, your IDL Project will display an icon with a red X through it to denote that it can no longer be found. For information on how to change the path of a missing file, see “Setting the Properties of a File” on page 476.

The IDL Project toolbar offers shortcuts to frequently used menu items. When you have a project open, the toolbar is available to help you manage your project's properties.

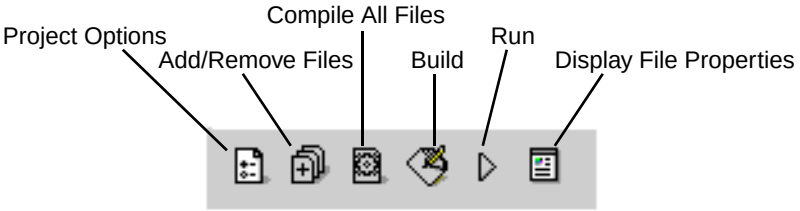


Figure 20-2: Project Toolbar

Example of a Project

A working example of a project, `demo_proj.prj`, has been included in the `examples` directory.

Where to Store the Files for a Project

The directory structure you use for your application files is an important consideration when you plan to export your application. It is important to create a directory structure which allows all files to be relative to the main project (.prj) file. Even though you can add any file from any path to your project, the following guidelines ensure that the application files will be found after you export your project.

1. **Create an organized directory structure containing all of your application files**. For example, you might create a directory structure similar to the following:

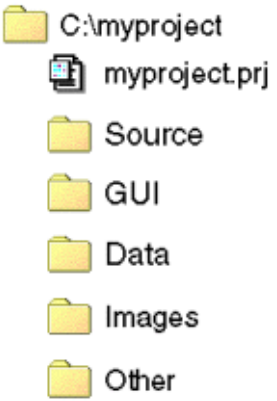


Figure 20-3: Example Directory Structure

Note

This example uses the same names as the default directory names displayed in the Project window. See "Modifying Project Groups" on page 469 for more information on the types of files stored in these groups. You do not have to name your directories in this manner. It is more important that all application files that you plan on exporting are organized in your local project directory.

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

When a project's files are exported, the files will be placed according to where they are in relation to the `.prj` file, keeping the directory structure intact whenever possible. All of the directories that are in the same directory as the `.prj` file will be recreated when an IDL Project is exported.

If you have files that are stored outside of this hierarchy, they will be exported to the top-level directory. If, for example, one of your files, `intertemp.dat`, exists in `D:\otherproj\data`, when you export your project it will be placed in the project's top-level directory as follows, `C:\myproject\intertemp.dat`. This may result in "File not found" errors when attempting to run your application after exporting it.

For more information on exporting a project, see "Exporting a Project" on page 488.

Creating a Project

To create a Project, complete the following steps:

1. Select **File** → **New** → **Project**. The **New Project** dialog is displayed.
2. Select the path and name of the project file. Click **Open** to create your project. A `.prj` extension will automatically be appended to the name you enter. You will see that your project appears in the **Project Window**
3. Save your new project. Select **File** → **Save Project**.

Note

You can only have one project open at a time. Before creating a new project, you must close any open projects.

After you have created your project, you'll see your project displayed in the Project Window. You will see that 5 groups have been automatically created when you created your project.

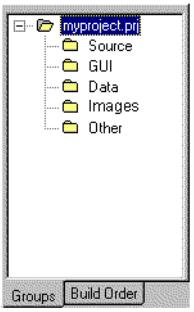


Figure 20-4: IDL Project Window

The following table describes the purpose for each group:

Group	Description
Source	Stores IDL source code files (`.pro`).
GUI	Stores GUI files (`.prc`) created using the IDL GUIBuilder.

Table 20-1: Project Group Descriptions

Group	Description
Data	Stores any data files.
Images	Stores image files.
Other	Stores any other files that do not apply to the other groups.

Table 20-1: (Continued) Project Group Descriptions

Opening, Closing, and Saving Projects

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

Opening a Project

To open a project, complete the following steps:

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

Tip

IDL keeps track of the most recently opened projects. You can use the **File → Recent Projects** menu to select a project to open.

Saving a Project

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

Closing a Project

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

Modifying Project Groups

After you have created your project, you can edit the groups for that project. You can create a new group or rename, remove, move up or down, or set to filter specific file types for the default groups.

Modifying Project Groups

To edit the groups in your project, complete the following steps:

1. Select **Project** → **Groups**. The Project Groups dialog is displayed:

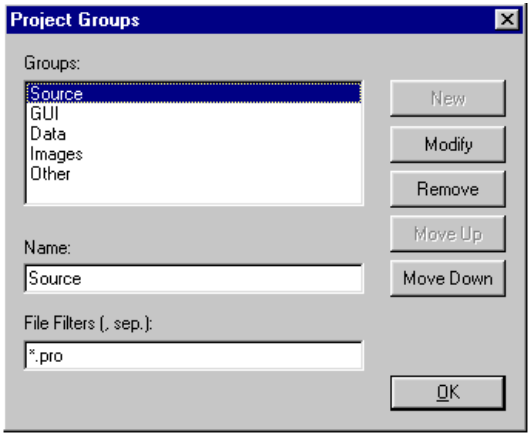


Figure 20-5: Project Groups Dialog

2. Through the Project Groups dialog, you can make the following changes:
   - **Create a New Group** — Enter a name into the Name text field and enter the desired file filter extensions, separated by commas, into the File Filters field. Click **New** to create the new group.
   - **Rename a Group** — Select the group that you want to rename. Edit the group name in the Name field and then click **Modify**.
   - **Move a Group** — Select a group listed in the Groups list and click **Move Up** or **Move Down**.
   - **Remove a Group** — Select the group you want to remove from the Groups list and click **Remove**.

- **Change the File Filter for a Group** — Enter file filter extension in the form *.*extension*. If you want more than one file type to be included in this group, separate each extension with a comma. For example, to include JPEG and PNG files, you would enter "`*.jpg, *.png`".

Note

When a file is added to a project, it is placed in the first group that meets the file extension criteria that is specified, with the first group being the uppermost group in the Groups list. If you have an all-inclusive filter (*), such as the "Other" group, you must place it at the bottom of the Groups list.

3. After you have completed making your changes, click **OK** to exit the Project Groups dialog.

Adding, Moving, and Removing Files

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

Adding Files

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

1. Open your project. Select **File** → **Open Project**. Select the path and name of your project file.
2. Click **Project** → **Add/Remove Files...** The **Add/Remove Files** dialog is displayed.

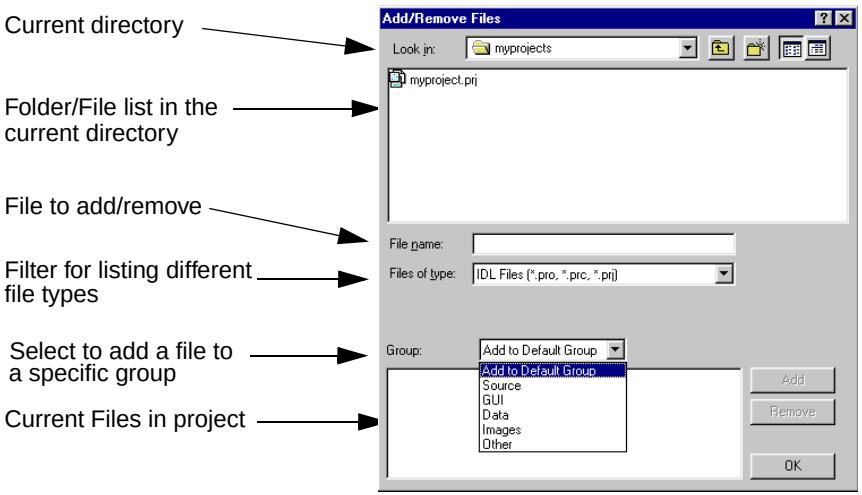


Figure 20-6: Add/Remove Dialog

3. Select the path and name of the file you want to add to your project. From the dropdown list, select the group you want to add the file to and click the **Add** button. You will see the file added to the list of current files in your project.
   - If your application contains an object that is defined in a `.pro` file, you must add the `.pro` file to your project list before building your project. If the object has any inherited properties from its superclass, you must also include the `.pro` file for the superclass if the superclass is a `.pro` file.

Objects using a `.pro` extension typically exist in the IDL distribution's `lib` subdirectory and its subdirectories. The *IDL Reference Guide* identifies object superclasses and gives the location of the object's source code.

For example, if you have defined an object in the file `myobject__define.pro`, and this object uses the methods of IDLgrLegend, you must include both `myobject__define.pro` and a copy of IDLgrLegend's source code, `idlgrlegend__define.pro`, in your project list.

If your object files call any other routines defined in `.pro` files, you must include these `.pro` files in your project list as well.

- If your application calls any routines via quoted strings, such as in CALL_PROCEDURE, CALL_FUNCTION, CALL_METHOD, EXECUTE, or in keywords that can contain procedure names such as TICKFORMAT or EVENT_PRO, you must include the `.pro` files for these routines in your project list.
- If your application uses IDL variables, such as a custom ASCII template, or if you want to distribute other procedures and functions that are not included in your main `.sav` file, you will need to create `.sav` files using the SAVE procedure. You must save variables and procedures in separate `.sav` files. These `.sav` files can be restored by using the RESTORE procedure in your main procedure, or can be restored automatically by IDL when resolving a routine with the same name as the `.sav` file.

Tip

You can also add files to your project by dragging and dropping the files from any file manager. (On some Motif platforms, dragging and dropping is not supported. In this case, use the **Add/Remove...** dialog.) If the file you want to add to your project is already open in an IDL editor window, right click in the editor window and select **Add to Current Project** from the shortcut menu.

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

Moving Files

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

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

Note

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

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

Note

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

Removing Files

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

To remove files from your project, complete the following steps:

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

Tip

You can use the shortcut menu to remove a file. Right click over the file and then select **Remove**. On Windows, you can also use the Delete key to remove files. Select the file by left-clicking over the file and then press the Delete key. On Motif, you can also highlight the file you want to remove, and press Ctrl+A to remove the file.

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

Working with Files in a Project

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

Editing a Source File

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

1. Open your project. Select **File** → **Open Project**. Select the path and name of your project file.
2. Access the shortcut menu by right-clicking over the file you want to open. Select **Edit** from the shortcut menu. Source files (`.pro`) are opened in the IDL editor and GUIBuilder files (`.prc`) are opened in the IDL GUIBuilder

Tip

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

Compiling a File

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

1. Open your project. Select **File** → **Open Project**. Select the path and name of your project file.
2. Access the shortcut menu by right-clicking over the file you want to compile. Select **Compile** from the shortcut menu. The file is compiled.

For more information on how to compile all the files in your project or just the files that have been recently modified, see “Compiling an Application from a Project” on page 484.

Testing a File

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

1. Open your project. Select **File** → **Open Project**. Select the path and name of your project file.
2. Access the shortcut menu by right-clicking over the file you want to test. Select **Test** from the shortcut menu. The file is run in test mode.

For more information on running `.prc` files in test mode, see "Running the Application in Test Mode" on page 607.

Tip

You can also compile and run IDL GUIBuilder files on any platform by building your project. For more information, see "Building a Project" on page 485.

Setting the Properties of a File

Each file in a project has properties. To view the properties of a file, access the shortcut menu by right-clicking over the file you want to test. Select **Properties** from the shortcut menu. Alternatively, you can select the file and click the **File Properties** toolbar button. The File Properties dialog appears as shown in the following figure.

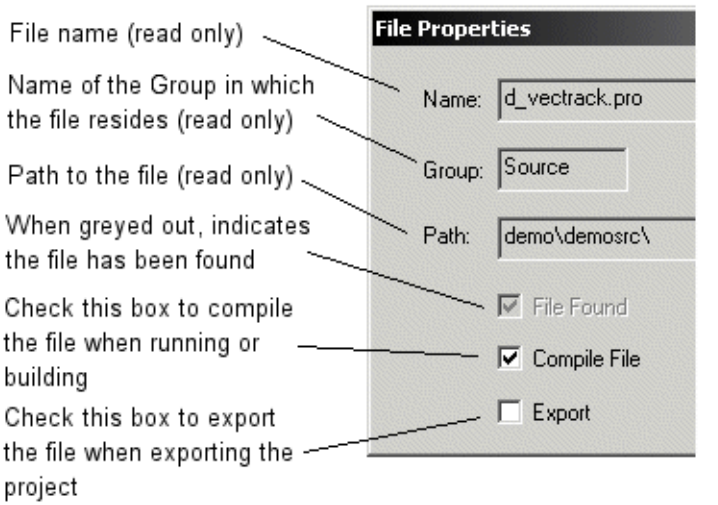


Figure 20-7: File Properties Dialog

The following table describes each property in detail:

Property	Description
File name	The name of the file. (This field is read only.)
Group	The name of the group in which the file resides. (This field is read only.)
Path	The path of the file. (This field is read only.)
File Found	This box appears grayed out when a file is found. If the file is not found, clicking on this checkbox displays a dialog so that you can specify the path of the file.
Compile File	Indicates whether or not to compile the file when running or building. For example, you may have included files for your main program that you do not want compiled. Leaving this check box blank indicates that you do not want this file compiled. **Note -** Non-source files such as data files and image files will be automatically excluded from compilation.
Export	Indicates whether or not to export the file when exporting a project. Some files, such as data files that you need to use when creating your application, are files that you do not want to export. When checked, this file will be exported.

Table 20-2: File Properties

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

Note

To set the properties of multiple files at a single time, see "Modifying Properties of Multiple Files" on page 478.

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

4. Select whether to compile the file. Check the **Compile File** checkbox to mark the file for compiling when running or building an application.
5. Select whether to export the file. You may select to export files such as data files if they are a necessary component of your application. Other data files which you have used for development but that aren't necessary need not be selected. Check the **Export** checkbox to export the file with your distribution. For information on arranging files for successful exporting, see"Where to Store the Files for a Project" on page 464.
6. Click **OK**.
7. Save your project file by selecting **File** → **Save Project**.

Modifying Properties of Multiple Files

To set the properties of a number of files at a single time, hold down CTRL and right-click to select multiple files in the Project window. Click the **File Properties** toolbar button. In the dialog which appears, you can select the **Compile File** or **Export** properties of "Multiple Files."

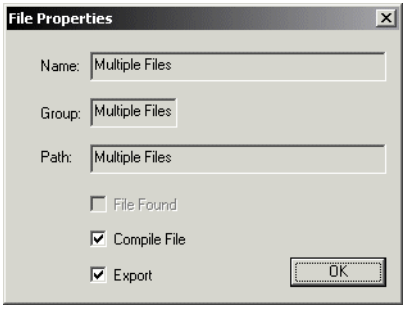


Figure 20-8: Multiple File Properties Dialog

In addition to setting file properties, you can also set the properties of your project. Through the Project Options dialog, you can control run and compile commands as well as selecting the type of project to create. See "Setting the Options for a Project" on page 479 for instructions.

Setting the Options for a Project

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

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

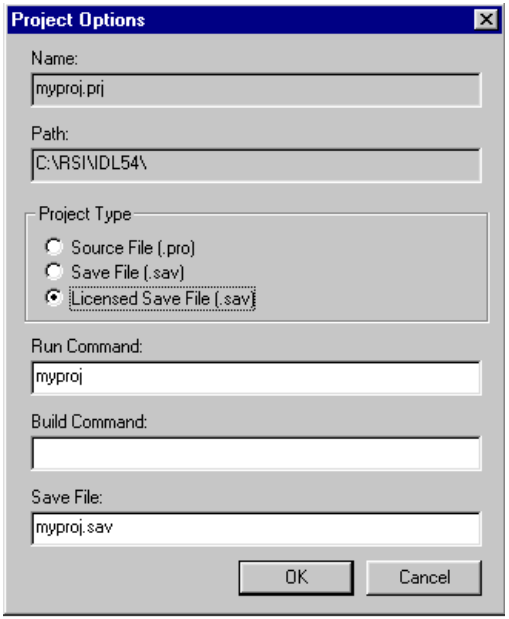


Figure 20-9: Project Options Dialog

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

Option	Description
Name	Specifies the project name. (This field is read only.)
Path	Specifies the path of the project. (This field is read only.)

Table 20-3: Project Options

Option	Description
Project Type	Specifies how the project will run or build. The available formats are: • Source File (`.pro`). • Save File (`.sav`). • Licensed Save File (`.sav`) **Note -** The Licensed Save File option is grayed out if you do not have an Unlimited Right to Distribute license. For more information on how to distribute IDL Runtime with your application, contact your RSI sales representative. For more information on building and running projects, see "Building a Project" on page 485 or "Running an Application from a Project" on page 487.
Run Command	Specifies the IDL command to run your application. The default is the name of the project. This can be any valid IDL command including `.sav` or `.pro` files (these can be files that are included or not included in your project.) Typically this is the main program in your application. **Tip -** You can use the %? command stream substitution to call a dialog to enter a value or values to pass to the called program. For example, if you have a program named "main" and it requires the argument "x" to be passed to it, then you can enter the following for the Run Command: `main, %?(Enter the value for x, x)` For more information on how to run your application, see "Running an Application from a Project" on page 487.

Table 20-3: (Continued) Project Options

Option	Description
Build Command	Specifies the IDL command to build the application. If left blank, the files in the project are built according to the Project Type specified and are compiled (if applicable) in the order specified under Build Order. For more information, see "Selecting the Build Order" on page 482. You can enter any valid IDL command including `.sav` or `.pro` files. You can also enter a batch file using *@filename* in order to perform other operations (for example, running a Perl script on your source or data files before compiling). For more information on batch scripts, see the *Using IDL* manual.
Save File	Specifies the name of the `.sav` file to create when building your project. For more information on building a project, see "Building a Project" on page 485. **Note -** This field is grayed out if you have selected the Source File (`.pro`) Project Type.

Table 20-3: (Continued) Project Options

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

Note

In addition to setting options for a project, you can also set an individual file's properties. For more information, see "Setting the Properties of a File" on page 476.

Selecting the Build Order

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

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

and file `AddTen.pro` contains:

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

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

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

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

using the scenario stated previously, the Build Order would look like the following:

Figure 20-10: Build Order Window

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

Note

If the Compile File option is deselected, the file will not show in the Build Order window. For more information on file properties, see "Setting the Properties of a File" on page 476.

Compiling an Application from a Project

You can compile all of the source files in your project, or just the files that you have recently modified. A modified file is one that has been modified and then saved. If you have included GUIBuilder files in your project, see the following section, "About IDL GUIBuilder Files".

Note

If you have dependencies on the order in which your files are compiled, see "Selecting the Build Order" on page 482.

To Compile All Files in Your Project

1. Open your project. Select **File → Open Project**. Select the path and name of your project file.
2. To compile all the files in your project, select **Project → Compile → All Files**.

To Compile Only Modified Files in Your Project

1. Open your project. Select **File → Open Project**. Select the path and name of your project file.
2. To compile just the files that have been modified since the last compilation, select **Project → Compile → Modified Files**.

Note

If you have dependencies on the order in which your files are compiled, see "Selecting the Build Order" on page 482.

Building a Project

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

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

  For more information on resetting an IDL session, see ".FULL_RESET_SESSION" in the *IDL Reference Guide* manual. For more information on resolving undefined but referenced functions, see "RESOLVE_ALL" in the *IDL Reference Guide* manual.

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

  The save file is created using the XDR and COMPRESS options. For more information, see "SAVE" in the *IDL Reference Guide* manual.

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

  For more information on how to create a licensed save file and distribute IDL Runtime with your application, contact your RSI sales representative.

Note

For more information on project options, see "Setting the Options for a Project" on page 479.

To build your project, complete the following steps:

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

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

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

3. Click **OK**.

Your project has been built.

About IDL GUIBuilder Files

When you build your IDL Project, the IDL GUIBuilder (`.prc`) files are automatically compiled and the resulting source (`.pro`) and event (`*_eventcb.pro`) files are automatically added to your project.

For more information on the IDL GUIBuilder, see Chapter 24, “Using the IDL GUIBuilder”.

Running an Application from a Project

After compiling your project, you can run your application. What happens when you run your project depends upon the project options you have selected:

- If you have selected your execution file format as source file, each file in your project is compiled and then run using the command you specified as the run command.
- If you have selected your execution file format as a `.sav` file, the most recently compiled version is run using the command you specified as the run command.

Note

You must have compiled or built your application before running it.

For more information on setting options for your project, see "Setting the Options for a Project" on page 479.

To run your application, complete the following steps:

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

Exporting a Project

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

What is exported is dependent upon the options you have selected for the project from the **Project** → **Options** dialog. If you have selected:

- **Source File** — Your project's source, IDL GuiBuilder, data, bitmaps, and any other files listed in your project will be exported along with your IDL Project file to a directory you specify so that you can move them to another platform. For information on how to set up a directory structure so that your IDL Project can find the source files after exporting, see "Where to Store the Files for a Project" on page 464.
- **Save File** — The `.sav` file for your project as well as data, bitmaps, and any other `.sav` files included in your project will be exported. You will also be given the option of exporting an IDL Runtime distribution for the platform to which you are exporting. Contact your sales person for options if you want to include an IDL Runtime distribution with your application. For information on how to set up a directory structure so that all files will retain their relative paths after exporting, see "Where to Store the Files for a Project" on page 464.
- **Licensed Save File** — The `.sav` file (with an embedded license) for your project as well as data, images, and any other `.sav` files included in your project will be exported. You will also be given the option of exporting an IDL Runtime distribution for the platform you are exporting on. For information on how to set up a directory structure so that all files will retain their relative paths after exporting, see "Where to Store the Files for a Project" on page 464.

  For more information on how to create a licensed save file and distribute IDL Runtime with your application, contact your RSI sales representative.

For more information on the options for a project, see "Setting the Options for a Project" on page 479.

Exporting Your Project's Source Files

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

1. Open your project. Select **File** → **Open Project**. Select the path and name of your project file.
2. Select "Source File (`.pro`)" or from the **Project** → **Options** dialog.
3. Select **Project** → **Export**. The **Browse for Folder** or **Export Directory** dialog is displayed.
4. Select the folder to which you want to export the project and click **OK**.

Your project is exported to the selected directory. When moving a project and its source files from one platform to another, there are a few items to be aware of:

- Project workspace information such as which files are open, etc. will not move from platform to platform.
- Problems with paths can occur if they are not relative paths. If you open a project and find that it cannot find the source file, you can fix this by changing the properties of the file. For more information, see "Where to Store the Files for a Project" on page 464 and "Setting the Properties of a File" on page 476.

Exporting Your Project to a Save File

To export your project to a save file, complete the following steps:

1. Open your project. Select **File** → **Open Project**. Select the path and name of your project file.
2. Select "Save File (`.sav`)" or "Licensed Save File (`.sav`)" from the **Project** → **Options** dialog.
3. Select **Project** → **Export**. The **Browse for Folder** or **Export Directory** dialog is displayed.
4. Select the folder to which you want to export the project and click **OK**.
5. A dialog is displayed asking if you want to export an IDL Runtime distribution with your `.sav` file. Select **No** to not include the distribution.

Your project is exported to the selected directory.

Exporting a Runtime Distribution

The process for exporting an IDL Runtime distribution of your project is slightly different depending on whether you run IDL on a Windows or UNIX platform.

Note

While a project exported with a Runtime distribution includes all of the support files necessary for a Runtime application, it does not include a Runtime license. If you are interested in including a Runtime version of IDL with your application, contact your RSI sales representatives for more information.

Under Microsoft Windows

Under Microsoft Windows, you can use the Project Export feature to create a complete runtime IDL distribution tree. To create a runtime distribution, do the following:

1. Open your project. Select **File** → **Open Project**. Select the path and name of your project file.
2. Select "Save File (`.sav`)" or "Licensed Save File (`.sav`)" from the **Project** → **Options** dialog.
3. Select **Project** → **Export**. The **Browse for Folder** dialog is displayed.
4. Select the folder to which you want to export the project and click **OK**. If the directory does not exist, you will need to create it: the Project editor will not create it for you.
5. A dialog is displayed asking if you want to export an IDL Runtime distribution with your `.sav` file. Select **Yes**.

6. The **Export Files** dialog appears, allowing you to select files to be included in the distribution.

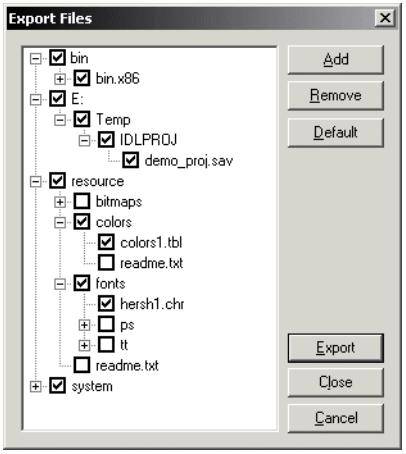


Figure 20-11: The Export Files dialog (Microsoft Windows only).

The Export Files dialog lists directories and files. Check marks indicate that a file is to be exported. The first time a project is exported, the files included in the `manifest_rt.txt` and `manifest_aux.txt` files are listed in the Export Files dialog. Only the files from `manifest_rt.txt` are checked for export.

- Click **Add** to select files to be added to the export list. Whether or not the files are actually exported depends on whether you check the checkbox next to the file's name.
- Click **Remove** to remove a file from the distribution to be exported.
- Click **Default** to restore the list of files to that are exported by default. By default, the files listed in the `manifest_rt.txt` file from the `\bin\make_rt` subdirectory of the IDL distribution are selected for export, and the files in the `manifest_aux.txt` file are displayed, but not selected for export.
- Click **Export** to export the files to the directory you specified step 4.
- Click **Close** to close the **Export Files** dialog without exporting the files. Your changes to the export list will be saved when you save the project file.
- Click **Cancel** to close the **Export Files** dialog, discarding any changes.

Your project, all of the selected IDL Runtime support files, and any other files you specified in the export list are exported to the directory you selected.

If an error is encountered during export, an Export Log detailing the export and the error is displayed in the bottom pane of the Export Files dialog.

Using the Export Feature without a Project (Windows Only)

On Microsoft Windows platforms, you can also use the Export feature without creating an IDL project file:

1. Select **Project → Export**. The **Browse for Folder** dialog is displayed.
2. Select the folder to which you want to export the project and click **OK**.
3. The **Export Files** dialog appears, allowing you to select files to be included in the distribution.
   - Click **Add** to select files to be added to the export list. Whether or not the files are actually exported depends on whether you check the checkbox next to the file's name.
   - Click **Remove** to remove a file from the distribution to be exported.
   - Click **Default** to restore the list of files to that are exported by default. By default, the files listed in the `manifest_rt.txt` file from the `\bin\make_rt` subdirectory of the IDL distribution are selected for export, and the files in the `manifest_aux.txt` file are displayed, but not selected for export.
   - Click **Export** to export the files to the directory you specified in step 2.
   - Click **Close** to close the **Export Files** dialog without exporting the files. Your changes to the export list will be saved when you save the project file.
   - Click **Cancel** to close the **Export Files** dialog, discarding any changes.

The files specified in the export list are exported to the directory you selected.

Under Unix

1. Open your project. Select **File → Open Project**. Select the path and name of your project file.
2. Select "Save File (`.sav`)" or "Licensed Save File (`.sav`)" from the **Project → Options** dialog.
3. Select **Project → Export**. The **Export Directory** dialog is displayed.
4. Select the directory to which you want to export the project and click **OK**.

5. A dialog is displayed asking if you want to export an IDL Runtime distribution with your `.sav` file. Select **Yes**. A complete list of the runtime distribution files that will be copied to your distribution directory is provided in the `manifest_rt.txt` text file. See Modifying the Manifest File for details on this file.

Your project is exported to the directory you selected.

Note

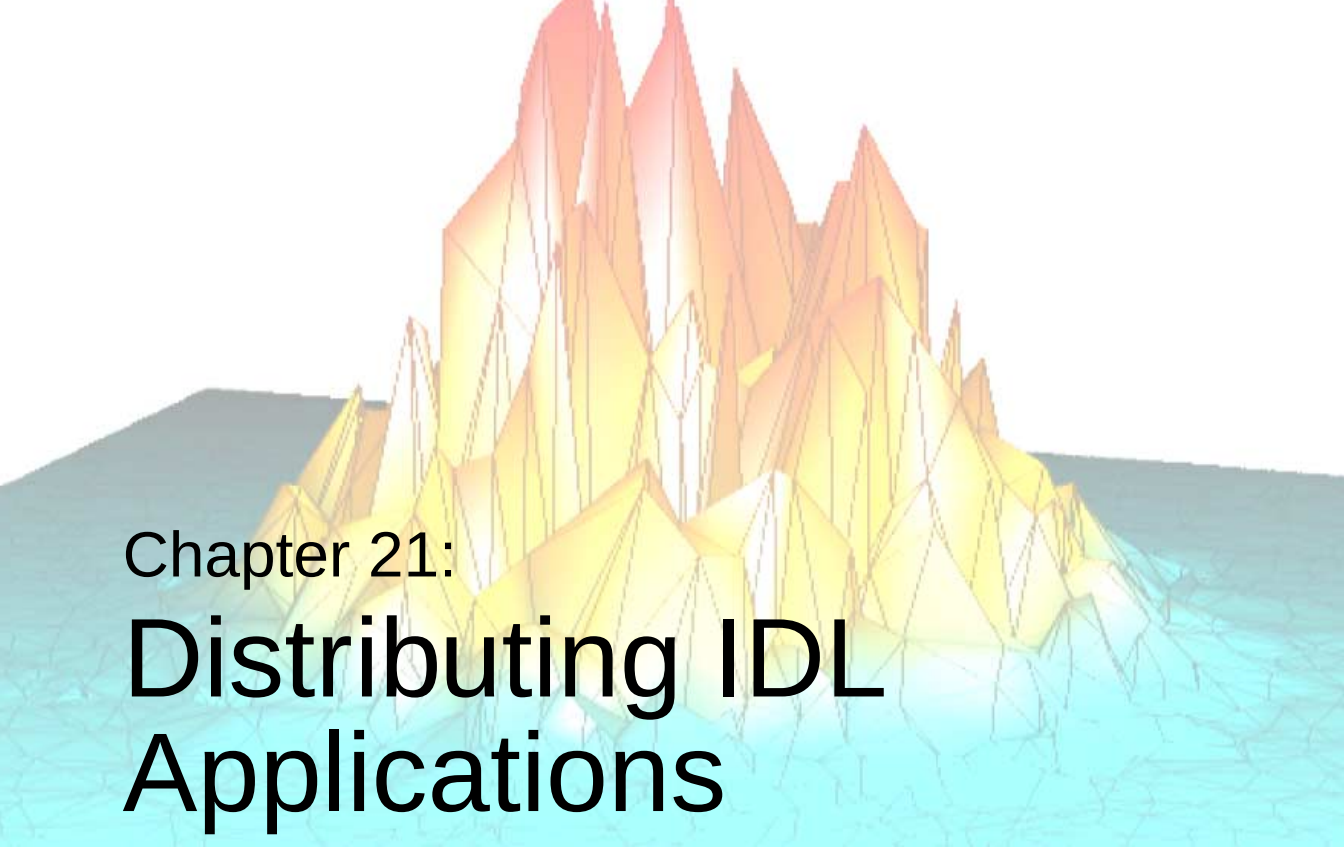
The **Project** → **Export** feature copies IDL binaries only for the platform from which you are currently running IDL. If you wish to create a distribution that supports multiple UNIX platforms, you must use the **Project** → **Export** feature to export a distribution for each platform you wish to support. You can specify the same destination directory each time you use the **Project** → **Export** feature, thereby creating a distribution with a `bin.`*`platform`* directory for each supported platform.

Modifying the Manifest File

The manifest file is located in *`idl-dir`*`/bin/manifest_rt.txt`, where *`idl-dir`* is the main IDL directory.

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

1. Open `manifest_rt.txt` in any text editor.
2. For an application that uses IDL DataMiner, copy the appropriate DataMiner files from `manifest_aux.txt`, (located in the same directory as `manifest_rt.txt`) to `manifest_rt.txt`.
3. Add the path and filename of any other files in the IDL distribution that you want to include to the list of files to export. Make sure that the path is relative to the *`idl-dir`*. Note that only IDL files can be added to the manifest.
4. Make sure that you have not included any blank lines in the file.
5. Save the file.



Chapter 21: Distributing IDL Applications

This chapter describes the following aspects of creating IDL applications for distribution:

What is a Stand-Alone IDL Application?

A *stand-alone IDL 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 IDL or the IDL Development Environment. All code written in IDL must be pre-compiled and provided in the binary `.sav` file format; no `.pro` files can be compiled by a stand-alone IDL application.

Note

If a stand-alone IDL application presents a user interface, it must be exposed via the IDL widget toolkit, since no access to the IDL command line or command output log is provided to the user.

Types of IDL Applications

IDL applications can either be written in IDL itself and distributed in IDL `.sav` files, or they can be written in another programming language and distributed in a compiled binary format. IDL applications fall into the following three broad categories:

- **Native IDL applications**. A native IDL application is written entirely in IDL and saved in a `.sav` file or series of `.sav` files that can be restored an run by an IDL distribution. While SAVE file applications can be run by fully-licensed copy of IDL, this chapter focuses on applications intended to be run as stand-alone applications by an IDL distribution in one of three more restricted modes:
  - in the IDL *Virtual Machine*.
  - with a *runtime* license.
  - with an *embedded* license.

  The process of creating applications written in IDL is the topic of this manual. This chapter describes the steps necessary to enable your application to run in one of the above modes when a full IDL license is not available.
- **Callable IDL applications**. An application that uses Callable IDL is an application written in another programming language, such as C or C++, that calls IDL as a subroutine. Callable IDL applications are run as standalone applications in one of the following modes:
  - with a *runtime* license.

- with an *embedded* license.

The process of creating Callable IDL applications is covered in the *External Development Guide*.This chapter describes the steps necessary to enable your application to run in one of the above modes when a full IDL license is not available.

- **IDL ActiveX Control applications**. The IDL ActiveX control can be used to access IDL functionality in applications written in other languages that support ActiveX, such as C++ or Visual Basic. IDL ActiveX applications are run as standalone applications in one of the following modes:
  - with a *runtime* license.
  - with an *embedded* license.

  The process of creating IDL ActiveX control applications is covered in the *External Development Guide*. This chapter describes the steps necessary to enable your application to run in one of the above modes when a full IDL license is not available.

Limitations of IDL Applications

IDL applications that run without a full IDL license — whether native IDL, Callable, or ActiveX — do not have access to the IDL compiler and have access to the IDL interpreter only in a restricted mode in which the provided commands are executed and the interpreter exits immediately. As a result, operations that require the compiler or that force the interpreter into an idle state will not execute when a full license is not present. In practice, this means that if you are writing an IDL application to be distributed to users who do not have a full IDL license, you should be aware of the following limitations:

Error Handling

Because the ON_ERROR procedure has the potential to force the IDL interpreter into an idle state when an error is encountered, it should not be used in IDL applications that will be distributed to users without a full IDL license. Use the CATCH procedure instead.

Blocking in Widget Applications

Similarly, blocking in the IDL widget event loop forces the IDL interpreter into an idle state. Because blocking behavior is the default in widget applications, be sure to use the NO_BLOCK keyword to XMANAGER if your IDL application includes a

user interface built from the IDL widget toolkit. See “XMANAGER and Blocking” in Chapter 26 for additional details.

Building a Native IDL Application

A native IDL application contains only IDL code — it does not contain code written in other programming languages such as C or Visual Basic — and can be exported as a `.sav` file for distribution to other IDL users. This section describes the process of packaging an application written entirely in IDL for distribution.

Example Native IDL Application

The following example illustrates how native IDL applications are developed and distributed.

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. Compile the Application

Select **Compile** from the Run menu to compile the `.pro` file.

3. RESOLVE_ALL

At the command line, enter the following to resolve all procedures and functions that are called in the application:

```
RESOLVE_ALL
```

4. SAVE

At the command line, enter the following to save the compiled application as a `.sav` file:

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

The resulting `.sav` file is a stand-alone IDL application that can be run on any Windows, UNIX or Mac OS X computer containing the IDL Virtual Machine or a licensed copy of IDL. If you want your customers to run this 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.

Licensing Options for IDL Applications

When you have an application that uses IDL, and you want to distribute it to users who do not already have IDL installed and licensed, you have two 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 RSI that enables you to bundle IDL with your application.

Free Runtime License (IDL Virtual Machine)

The IDL Virtual Machine is a runtime version of IDL that can execute IDL `.sav` 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 "The IDL Virtual Machine" on page 503 for additional details.

Purchased Licenses

When you purchase a license from RSI, you can include a licensed IDL installation with your IDL application, Callable IDL application, or IDL ActiveX Control application. There are two types of 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 Development Environment (IDLDE), 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 may choose to distribute your

application with a fully-licensed copy of IDL. Contact your sales representative to purchase copies that you can distribute.

Embedded Licenses

Embedded licenses must be purchased from RSI. Purchasing an embedded license enables you to build runtime license information into IDL `.sav` files, callable IDL applications, or IDL ActiveX applications.

Runtime Licenses

Runtime licenses must also be purchased from RSI. A runtime license enables a single user to run IDL `.sav` files, callable IDL applications, or IDL ActiveX applications. Runtime licenses come in two varieties:

- node-locked FlexLM licenses that can be installed only on a single machine,
- licenses that can be installed on any machine of a specified platform.

Note

Licenses that are not node-locked are more expensive than node-locked licenses.

The IDL Virtual Machine

The IDL Virtual Machine 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 `.sav` 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. During installation, you can choose to install just the IDL Virtual Machine or a full installation of IDL (which includes the IDL Virtual Machine). 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 `.sav` file program is run without an IDL license, IDL defaults to the IDL Virtual Machine mode.

To distribute an application for the IDL Virtual Machine, follow these steps:

1. Build your IDL application. See "Building an Application that Runs in the IDL Virtual Machine" on page 504.
2. Provide your users with instructions for installing the IDL Virtual Machine. See the *Installing and Licensing IDL 6.0* manual.
3. Provide your users with instructions for running your IDL application in the IDL Virtual Machine. See "Running a .sav File in the IDL Virtual Machine" on page 505.
4. Distribute your application.

Limitations of Applications that Run in the IDL Virtual Machine

The IDL Virtual Machine will run a compiled IDL `.sav` file even if no IDL license is present. RSI's aim with the IDL Virtual Machine is to facilitate IDL code collaboration and application distribution. However, a few restrictions exist:

- The IDL Virtual Machine displays a splash screen on startup.
- `.sav` files must be created using IDL version 6.0 or later.
- No access to the IDL command line or IDL compiler is provided.

- 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.)
- Callable IDL applications and applications that use the IDL ActiveX control will not run in the IDL Virtual Machine.
- The IDL Virtual Machine must be installed via the installation program provided by RSI. You are prohibited from modifying the IDL Virtual Machine distribution.

Note

The IDL Virtual Machine installation program does not install the IDL high resolution maps. If your application uses the high resolution maps, users must install the full version of IDL rather than the Virtual Machine installation.

Building an Application that Runs in the IDL Virtual Machine

An IDL program compiled in IDL 6.0 or later that does not use the EXECUTE function can be saved as a `.sav` file that will run in the IDL Virtual Machine. If a program is to be run in the IDL Virtual Machine, it is not necessary to include an IDL distribution with the `.sav` file because IDL Virtual Machine is installed on the user's machine. The `.sav` file need only include your own code, creating a smaller file that is easier to distribute.

To create `.sav` files to run in the IDL Virtual Machine, do one of the following:

- Create `.sav` files from one or more compiled `.pro` files with the SAVE procedure. For instructions, see"Using SAVE and RESTORE" on page 522. Refer to "Saving Compiled IDL Programs" in Chapter 9 for details, and refer to "SAVE" in the *IDL Reference Guide* manual.
- Create `.sav` files from a project by selecting **Project → Export** with the **Save File (`.sav`)** option specified. For instructions, see "Using Project Æ Export" on page 524, and refer to Chapter 20, "Creating IDL Projects" for details.
- (UNIX only) Create a distribution with the `make_rt` script. For instructions, see "Using the make_rt Script (UNIX only)" on page 527.

Note

Creating `.sav` files of object-oriented programs requires the use of RESOLVE_ALL with the CLASS keyword.

Version Compatibility of .sav Files

The IDL Virtual Machine will execute IDL routines stored in `.sav` files created with IDL version 6.0 and later. Any `.sav` files created with previous versions of IDL must be recompiled using IDL 6.0 or later for them to run with the Virtual Machine.

Running a .sav File in the IDL Virtual Machine

How you run a `.sav` file in the IDL Virtual Machine depends on your operating system:

Windows

Windows users can drag and drop the `.sav` file onto the IDL Virtual Machine desktop icon, launch the IDL Virtual Machine and open the `.sav` file, or launch the `.sav` file in the IDL Virtual Machine from the command line.

To use drag and drop:

1. Locate and select the `.sav` file in Windows Explorer.
2. Drag the file icon from the Windows Explorer list and drop it onto the **IDL Virtual Machine 6.0** icon that has been created for you on the desktop.
3. Click anywhere in the window to dismiss the IDL Virtual Machine splash screen and run the `.sav` file.

To open a `.sav` file from the IDL Virtual Machine:

1. Do either of the following to launch the IDL Virtual Machine and display the IDL Virtual Machine window:
   - Select **Start → Programs → RSI IDL 6.0 → IDL Virtual Machine** or **Start → Programs → RSI IDL Virtual Machine 6.0 → IDL**
   - Double-click the **IDL Virtual Machine 6.0** desktop icon.
2. Click anywhere in the window to dismiss the IDL Virtual Machine splash screen and display the file selection menu.
3. Locate and select the `.sav` file, and double-click or click **Open** to run it.

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 directories to the *RSI-DIR*`\bin\bin.x86` 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. If you omit the path and filename (leaving only the `-vm` flag), IDL will open a file selection dialog allowing you to select the `.sav` file.

Note

If a license is available on the machine running the `.sav` file, double-clicking the `.sav` file will run it in the licensed version of IDL. To force the `.sav` file to run in the Virtual Machine, either drag and drop the `.sav` file on the IDL Virtual Machine icon or run it from the command line with the `-vm` argument.

UNIX and Mac OS X

UNIX and Mac OS X 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>.sav
   ```

   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 window to dismiss the IDL Virtual Machine splash screen 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 dismiss the IDL Virtual Machine splash screen and display the file selection menu.

3. Locate and select the `.sav` file, and click **Open**.

Example

To distribute the example IDL application as a `.sav` file to be run in the IDL Virtual Machine:

1. Build the application and create the `myApp.sav` file. See "Example Native IDL Application" on page 499.
2. Distribute the `myApp.sav` file to your users, providing instructions for them to install the IDL Virtual Machine and run myApp.sav in the Virtual Machine.

Embedded Licensing

An *Embedded* license allows you to distribute copies of IDL licensed to run only your application to multiple users. Licensing an IDL application with an embedded license is the simplest form of licensing. When you purchase an embedded license, your version of IDL is enabled with the ability to automatically embed a license in your application `.sav` file when you build your project by selecting the **Licensed Save File** option from the IDLDE **Project** → **Options** dialog.

To distribute an application with an embedded licence, follow these steps:

1. Obtain and install your embedded license. How you do this depends on the type of application you are licensing. See "Licensing a Native IDL Application" on page 508, "Licensing a Callable IDL Application" on page 509, or "Licensing an IDL ActiveX Application" on page 511.
2. Build your application. See "Building Your Application" on page 522.
3. Prepare your distribution. See "Preparing a Distribution" on page 531.
4. Provide the means for installing your application. See "Installing your Application" on page 538.
5. Distribute your application.

Including License Information in your Application

How you include licensing information in your application depends on the type of application you have created.

Tip

To ensure that your application will run with your embedded license and not in the IDL Virtual Machine, add the following code to your application before preparing your application distribution:

```
a = lmgr(/vm)
if a then begin
   b = DIALOG_MESSAGE('Please contact the author for
   licensing instructions')
   return
endif
```

Licensing a Native IDL Application

To license a native IDL application with an embedded license, do the following:

1. **Obtain and Install an Embedded License.** Contact RSI to purchase your license. Use the Licensing Wizard to install your new license. This will enable the **Licensed Save File (.sav)** option in the IDL Project Options dialog.
2. **Create an IDL Distribution**. Create an IDL distribution for your application by following the steps described in "Building Your Application" on page 522.

Licensing a Callable IDL Application

Note

It is beyond the scope of this manual to discuss the creation of Callable IDL applications. See Chapter 21, "Callable IDL" in the *External Development Guide* for details. Note that applications using an embedded license must set the **IDL_INIT_EMBEDDED** option when calling the **IDL_Init()** or **IDL_Win32Init()** function, and must call **IDL_RuntimeExec()** rather than **IDL_Exec()**.

Licensing a Callable IDL application consists of embedding a license string along with some initialization code into your application code before your initial call to IDL. The license information will be provided to you by RSI.

1. **Obtain Your Licensing Information**. Contact RSI 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 or ActiveX).
   - Calling program language (i.e., VB, C++, C, Fortran).

   You will receive a text file containing a function that IDL uses to retrieve the licensing information.
2. **Modify Your Application Code**. After you receive your license information, make the following changes to your application code. These instructions assume your code is written in C.
   A. After you receive your license information, place the function that IDL uses to retrieve licensing information in the module from which you are

initializing IDL. Although your licensing information is individualized, it will resemble the following:

```
/* Callable Application license for: myapp, My App */
/* License built for IDL Version 6.0 */
char ** IDL_STDCALL callAppLicFunc() {
static char *initStr[] = {
 "12345678abcdabcd",
 "12345678abcdabcd",
 "12345678abcdabcd",
 "12345678abcdabcd",
 "12345678abcdabcd",
 "12345678abcdabcd",
 "12345678abcdabcd",
 "12345678abcdabcd",
 "12345678abcdabcd",
 "12345678abcdabcd",
 "12345678abcdabcd",
 "12345678abcdabcd",
 "12345678abcdabcd",
 "12345678abcdabcd",
 "12345678abcdabcd" };

return (initStr);
}
```

B. Declare the following struct in the module from which you are initializing IDL. This is used by both IDL and the callable application, but isn't exposed to the user:

```
typedef struct _callAppLicInfo{
    unsigned long   dwKey;
    char ** (IDL_STDCALL *callAppLicFunc)();
} CALLAPPLICINFO;
```

C. Allocate the struct before the initializing IDL.

```
CALLAPPLICINFO      callAppLicInfo;
```

D. Initialize the struct:

```
callAppLicInfo.dwKey = 0xCA00CA00;
callAppLicInfo.callAppLicFunc = callAppLicFunc;
```

E. Initialize IDL with one of the following statements:

For UNIX and Macintosh:

```
if (IDL_Init_CallAppLicense(0, &callAppLicInfo))
```

For Windows:

```
if (!IDL_Win32Init(0, hInstance, hwnd, &callAppLicInfo))
    return(IDL_FALSE);
```

3. **Create an IDL Distribution**. If you have not yet created an IDL distribution for your application, do so now by following the steps described in "Building Your Application" on page 522. If you have already created a distribution, add your modified executable to the distribution. Note that if your Callable IDL application uses a `.sav` file, it does not matter whether you choose the **Licensed Save File** or **Save File** option in the Project Options dialog when building your project because the licensing information is embedded in your application code rather than the `.sav` file. See "Using Project Æ Export" on page 524.
4. Your application can now be prepared for distribution by following the steps in "Preparing a Distribution" on page 531.

Licensing an IDL ActiveX Application

Licensing an IDL ActiveX control application consists of embedding an IDL initialization string into your application code before your initial call to IDL. The license information will be provided to you by RSI.

1. **Obtain Your Licensing Information**. Contact RSI for your license information. You will need to provide the following information:
   - The license installation number for your embedded license. Note that this 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 or ActiveX).
   - Calling program language (i.e., VB, C++, C, Fortran).

   You will receive a text file containing an initialization string.
2. **Modify Your Application Code**. After you receive your license information, insert the initialization string into your code prior to calling IDL. Although the licensing information you receive will be slightly different, it will resemble the following:

```
' IDL ActiveX Control Application license for: myapp, My App
' License built for IDL Version 6.0
theApp.InitStringInfo("12345678abcdabcd, -
     12345678abcdabcd, _
```

```
12345678abcdabcd, _
12345678abcdabcd, _
12345678abcdabcd, _
12345678abcdabcd, _
12345678abcdabcd, _
12345678abcdabcd, _
12345678abcdabcd, _
12345678abcdabcd, _
12345678abcdabcd, _
12345678abcdabcd, _
12345678abcdabcd, _
12345678abcdabcd")
```

Note

The `InitStringInfo` method must be called prior to ActiveX initialization.

3. **Create an IDL Distribution**. If you have not yet created an IDL distribution for your application, do so now by following the steps described in Chapter 21, "Building Your Application". If you have already created a distribution, add your modified executable to the distribution. Note that if your ActiveX application uses a `.sav` file, it does not matter whether you choose the **Licensed Save File** or **Save File** option in the Project Options dialog when building your project because the licensing information is embedded in your application code rather than the `.sav` file. See "Using Project Æ Export" on page 524.

Your application can now be prepared for distribution by following the steps in Chapter 21, "Preparing a Distribution".

Running a .sav File with an Embedded License

How you run a `.sav` file with an embedded license depends on your operating system:

Windows

Do either of the following:

- If your application was built in the IDL Project Editor, find the `.exe` file in the distribution folder and double-click the filename. See "Starting Your Windows Application" on page 532
- From the command line prompt, change directories to the *RSI-DIR*`\bin\bin.x86` directory and enter the following:

```
idlrt -em=<path><filename>
```

where <*path*> is the path to the `.sav` file, and <*filename*> is the name of the `.sav` file.

UNIX or Mac OS X

At the UNIX command prompt, enter the following:

```
idl -em=<path><filename>
```

where <*path*> is the path to the `.sav` file, and <*filename*> is the name of the `.sav` file.

Note

If you set the IDL_DIR environment variable, you can simply execute the startup script. See "Starting Your UNIX Application" on page 536.

Example

To distribute the example IDL application with an embedded license:

1. Request your embedded license and install it on your development machine.
2. Build your application in the IDL Project Editor. Create a new project, add the `myApp.pro` and `imagefile.sav` files (see "Example Native IDL Application" on page 499) to the project, and **Export** your project with the **Licensed save file (.sav)** project option selected (see "Using Project Æ Export" on page 524 for details). The resulting `myApp.sav` file will include a licensed runtime distribution of IDL.
3. Distribute `myApp.sav` and the runtime distribution files to your users, providing instructions on how to run myApp.sav in the runtime distribution.

Runtime Licensing

A *Runtime* license allows you to deliver a copy of IDL that is licensed to run only your application on a single machine. 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 FLEXlm license tied to the specific machine on which your application will run (so you will need to obtain information about your customer's machine), or a more costly but less restricted license that will run on any machine of a given platform.

You can choose how to license your application:

- 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 you must 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.

To distribute an application with a runtime license, follow these steps:

1. Build your application. How you do this depends on the type of application you are building. See "Building a Native IDL Application" on page 515, "Building a Callable IDL Application" on page 515, or "Building an IDL ActiveX Application" on page 516. For instructions on using the tools available for building applications, see "Building Your Application" on page 522
2. Obtain and install your runtime license. See "Obtaining and Installing a Runtime License" on page 516.
3. Prepare your distribution. See "Preparing a Distribution" on page 531.
4. Provide the means for installing your application. See "Installing your Application" on page 538.
5. Distribute your application.

Building a Runtime Application

The procedure for building an application to run with a runtime license depends on the type of application.

Tip

To ensure that your application will run with your runtime license and not in the IDL Virtual Machine, add the following code to your application before preparing your application distribution:

```
a = lmgr(/vm)
if a then begin
   b = DIALOG_MESSAGE('Please contact the author for
   licensing instructions')
   return
endif
```

Building a Native IDL Application

To build a native IDL application with a runtime license, create an IDL distribution for your application by following the steps described in "Building Your Application" on page 522. Keep the following in mind when creating the distribution:

- If you use the **Project → Export** feature, make sure to select the **Save File (.sav)** option in the Project Options dialog before building your project. See "Build Your Project" on page 525.
- If you use the `make_rt` script, you will need to create your `.sav` file manually, and make sure to specify "rt" for the mode parameter of the `make_rt` command. See "Syntax of the make_rt Script" on page 529.

Building a Callable IDL Application

Note

It is beyond the scope of this manual to discuss the creation of Callable IDL applications. See Chapter 21, "Callable IDL" in the *External Development Guide* for details. Note that applications using a runtime license must set the **IDL_INIT_RUNTIME** option when calling the **IDL_Init()** or **IDL_Win32Init()** function, and must call **IDL_RuntimeExec()** rather than **IDL_Exec()**.

To license a Callable IDL application with a runtime license, create an IDL distribution for your application by following the steps described in "Building Your

Application” on page 522. If your Callable IDL application uses a `.sav` file, keep the following in mind when creating the distribution:

- If you use the **Project → Export** feature, make sure to select the **Save File (.sav)** option in the Project Options dialog before building your project. See “Build Your Project” on page 525.
- If you use the `make_rt` script, you will need to create your `.sav` file manually, and make sure to set the `savefile` parameter to the name of your application executable. See “Syntax of the make_rt Script” on page 529.

Building an IDL ActiveX Application

To license an IDL ActiveX application with a runtime license, create an IDL distribution for your application by following the steps described in “Preparing a Distribution” on page 531. If your ActiveX application uses a `.sav` file, keep the following in mind when creating the distribution:

- If you are using the **Project → Export** feature, make sure you have selected the Save File (.sav) option in the Project Options dialog when building your project. See “Build Your Project” on page 525.
- If you are using the `make_rt` script, you will need to create your `.sav` file manually, and make sure to set the `savefile` parameter to the name of your application executable. See “Syntax of the make_rt Script” on page 529.

Obtaining and Installing a Runtime License

Runtime applications that run on either Windows or UNIX platforms are licensed using either node-locked (FLEXlm) licenses or non-node-locked single-user licenses. Node-locked licenses are tied to the specific machine on which the application will run, while non-node-locked licenses will run on any machine of a given type. The following is an overview of the process you will follow to license your runtime application on either Windows or UNIX:

1. Get information about the end user’s machine on which your application will run.
2. Send this information to RSI. A license file will be generated and sent to you.
3. Install the license file in your distribution to create a licensed distribution so that the end user can install and run your application without the need for any changes. Alternatively, if you have created a single unlicensed distribution that

you provide to all your end users, you can provide the end user with a separate license file and instructions for installing the license file.

The following sections describe these steps in detail for each platform.

Windows

Obtaining a 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 is to be licensed. If your end user has already installed an unlicensed copy of your application, he or she will have access to `lmtools.exe`. Otherwise, you will need to provide the end user with a copy the `lmtools.exe` file, which can be found in the `bin/bin.x86` directory of your IDL distribution.

Note

If you are using a non-node-locked license, you only need to know which platform (Windows, Unix) your end user will be using.

Provide the 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 configured network interface card.
2. Run the `lmtools.exe` application. The Lmtools dialog appears.
3. Click the **Hostid** button. Information similar to the following is displayed:

```
Hostd ID's---------------------------------
HOSTNAME=myhost
USER=jdoe
DISPLAY=myhost
INTERNET=10.15.2.109
0030dcb86317
DISK_SERIAL_NUM=c5f8b462
```

4. Click the **Save Text** button. In the Save As dialog, enter a path a filename and click **Save**.
5. Send the text file saved in the previous step to your application vendor.

When your end user has provided you with the information obtained by `lmtools.exe`, e-mail this information to `register@RSInc.com` or fax the information to RSI at (303) 786-9909. If you did not purchase IDL directly from RSI, send the file to your local distributor.

RSI will then send you a license file called `license.dat`.

Installing the License

Once you have received a `license.dat` file from RSI, perform the following steps to provide your end user with a licensed copy of your application. If you have provided an unlicensed copy of your application and want the end user to license the application, provide the end user with the following instructions:

Note

If the LM_LICENSE_FILE environment variable has been set on the user's system, the location of the license file specified in the `idl.ini` file will be ignored. In this case, the user will either need to add the license file path to the existing LM_LICENSE_FILE value, or move the license file to the location specified by the existing LM_LICENSE_FILE value.

1. Edit the `idl.ini` file using the instructions under "Edit the idl.ini File" on page 531.
2. Place the `license.dat` file in the location specified by the `idl.ini` file. For example, assume the application directory hierarchy of your distribution looks like this:

   ```
   MyApp
      bin
         bin.x86
      resource
   ```

   Assume the `idl.ini` file is located in the `MyApp\bin\bin.x86` directory along with your application executable, and the `idl.ini` file contains the following line:

   ```
   RSI Root=..\..\..
   ```

   For this RSI Root value, you must create a `license` directory at the same level as the `MyApp` directory, as shown below:

   ```
   MyApp
      bin
         bin.x86
      resource
      license
   ```

   Then place the license file in the `license` directory.

   Note

   All directories specified in the `idl.ini` file should be relative to the directory containing the `.ini` file. This ensures that, if the directory tree for your application is moved to another location, it will still run.

3. You should now be able to run the application by double-clicking the application `.exe` file, located in the `bin\bin.x86` subdirectory of the application distribution.

UNIX

Obtaining a 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 is to be licensed. If your end user has already installed an unlicensed copy of your application, he or she will have access to `lmhostid`. Otherwise, you will need to provide the end user with a copy the `lmhostid` file, which can be found in the `bin` directory of your IDL distribution.

Note

If you are using a non-node-locked license, you only need to know which platform (Windows or Unix) your end user will be using.

Provide the end user with the following instructions:

1. In the `bin` directory of the application distribution, execute the command `lmhostid`. 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@RSInc.com` or fax the information to RSI at (303) 786-9909. If you did not purchase IDL directly from RSI, send the file to your local distributor.

RSI will then send you a license file called `license.dat`.

Installing the License

Once you have received a `license.dat` file from RSI, perform the following steps to provide your end user with a licensed copy of your application. If you have provided an unlicensed copy of your application, and want the end user to license the application, provide him or her with the following instructions:

1. Modify the LM_LICENSE_FILE environment variable on the machine on which the application will run. This can be performed manually by the end user, or can be performed automatically using an installation script. If there is

no LM_LICENSE_FILE environment variable defined, simply set the value of LM_LICENSE_FILE to the desired directory, such as in the following command:

```
setenv LM_LICENSE_FILE /myapp/license/license.dat
```

If there is already an existing value for LM_LICENSE_FILE, append the path for the license directory of your application to the existing LM_LICENSE_FILE value.Separate the new license path from the existing one with a colon as follows:

For C shell:

```
setenv LM_LICENSE_FILE /home/otherapp/license.dat:
   /myapp/license/license.dat
```

2. Place the `license.dat` file in directory specified by the LM_LICENSE_FILE environment variable. For example, assume the application directory hierarchy of your distribution looks like this:

```
myapp
   bin
      bin.sgi
   license
   resource
```

If the path `/myapp/license` has been appended to the end user's LM_LICENSE_FILE environment variable, IDL will look for the `license.dat` file in the `/myapp/license` directory.

Running a .sav File with a Runtime License

How you run a `.sav` file with a runtime license depends on your operating system:

Windows

Do one of the following:

- If your application was built in the IDL Project Editor, find the `.exe` file in the distribution folder and double-click the filename. See "Starting Your Windows Application" on page 532
- From the command line prompt, change directories to the *RSI-DIR*`\bin\bin.x86` directory and enter the following:

  ```
  idlrt <path><filename>
  ```

  where <*path*> is the path to the `.sav` file, and <*filename*> is the name of the `.sav` file.

UNIX or Mac OS X

At the UNIX command prompt, enter the following:

```
idl -rt=<path><filename>
```

where <*path*> is the path to the `.sav` file, and <*filename*> is the name of the `.sav` file.

Note

If you set the IDL_DIR environment variable, you can simply execute the startup script. See "Starting Your UNIX Application" on page 536.

Example

To distribute the example IDL application with a runtime license:

1. Build the application and create the `myApp.sav` file. See "Example Native IDL Application" on page 499.
2. Obtain licensing information from each of your customers by having them run `lmtools.exe` (Windows) or `lmhostid` (UNIX), and send the information to RSI to generate a license file for each customer.
3. When you receive the license files, do either of the following:
   - Include a license file with each distribution of the application, send it to the appropriate customer, and provide instructions for the customer to install and license the application.
   - Install a license file in the distribution, generate a separate licensed distribution for that customer, and repeat for each subsequent license file. Each customer receives a custom licensed distribution.
4. Distribute your application with the appropriate instructions.

Building Your Application

This section discusses the process of creating an application distribution that includes the files necessary to run IDL, thereby allowing you to distribute your application to users who do not already have IDL installed. The following is an overview of the process of creating an IDL distribution:

1. Decide whether you will use the SAVE and RESTORE procedures, the IDLDE **Project → Export** feature, or the `make_rt` script to create the distribution. In choosing which method you will use, consider the following guidelines:
   - For IDL applications, as well as Callable IDL and ActiveX applications that restore IDL `.sav` files, you may wish to use the **Project → Export** feature to create the distribution. The IDL Projects interface automates the process of creating the `.sav` file for your application.
   - UNIX Users: For Callable IDL and ActiveX applications that do not use a `.sav` file, you may wish to use the `make_rt` script to create your distribution. Because the **Project → Export** feature requires you to create a project and build a `.sav` file before you export an IDL distribution, it is easier to use the `make_rt` script when your application does not use a `.sav` file. (The `make_rt` script is not available in IDL for Windows.)
2. Create the distribution using SAVE, the **Project → Export** feature, or the UNIX `make_rt` script. See Using SAVE and RESTORE, "Using Project Æ Export" on page 524, or "Using the make_rt Script (UNIX only)" on page 527.

Using SAVE and RESTORE

The SAVE procedure allows you to save IDL variables, system variables, and IDL functions and procedures as `.sav` files. This section explains when to use the SAVE procedure to create `.sav` files, when to use the RESTORE procedure to restore these `.sav` files in your application, and when it is or is not necessary to use RESTORE to explicitly restore your `.sav` files. For instructions on using SAVE and RESTORE, see "Saving Compiled IDL Programs" in Chapter 9, and refer to "SAVE" in the *IDL Reference Guide* manual for details.

When To Use SAVE

For distributable applications, IDL does not compile `.pro` files. Therefore, any procedures or functions used by your application must be resolved and contained in a `.sav` file. For IDL applications, these routines can be part of the main `.sav` file that is restored when your application is started. If you use an IDL project to create your distribution and add the required `.pro` files to your project before it is built, you do

not need to manually create .sav files with the SAVE and RESOLVE_ALL procedures. The following are examples of cases in which you might use SAVE to create .sav files:

- To create .sav files for any procedures or functions that are not contained in the main .sav file that is restored when a native IDL application is started.
- To create .sav files for any procedures or functions used by a Callable IDL or ActiveX application.
- To create .sav files for any variables used by your application, such as custom ASCII templates.

Note

A single .sav file cannot contain both routines and variables.

When To Use RESTORE

There are three ways to restore a .sav file:

- Explicitly restore the .sav file using RESTORE. This requires you to specify the path to the .sav file.
- Call the procedure with the same name as the .sav file. 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 it automatically.
- Specify the filename of the .sav file as the argument to the IDL -vm command, as follows:

```
UNIX: idl -vm =<filename>
Windows: idlrt -vm=<filename>
```

Because calling a procedure with the same name as a .sav file allows IDL to automatically find and restore the .sav 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 .sav file containing variable data.
- When your .sav 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 .sav 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.

Single vs. Multiple .sav Files

There are several ways to include the necessary routines in your application:

- For a native IDL application, include all routines in the main `.sav` file that is restored when your application is started. This makes all routines available without having to restore any additional `.sav` files, and reduces the number of `.sav` 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 `.sav` file containing all your routines. You might use this method for a Callable or ActiveX application, if you want to keep certain routines separate from your main `.sav` file in a native IDL application, or if your application includes routines provided to you as a `.sav` file by another developer. To run any routines included in this `.sav` file, you must restore the `.sav` file by either calling a routine with the same name as the `.sav` file or restore it explicitly using RESTORE.
- Create a separate `.sav` 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 `.sav` 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.

Examples

See “Creating a .sav File of a Simple Routine” on page 203 and “Customizing and Saving an ASCII Template” on page 204 for examples.

Using Project → Export

This section describes how to export an IDL distribution using the **Project → Export** option on the IDLDE menu. Exporting a project is essentially a three-step procedure:

1. Create your project. See “Before Building Your Project” on page 525 and “Add Required Files to Your Project” on page 525.
2. Build your project. See “Build Your Project” on page 525.
3. Export your project. How you do this depends on your operating system. See “Exporting Your Project in Windows” on page 527 or “Exporting Your Project in UNIX” on page 527.

For complete information on building and exporting projects, see Chapter 20, “Creating IDL Projects”.

Before Building Your Project

Before building your project, check your application for the following:

1. Verify the procedure name in your main `.sav` file. The main `.sav` file to be executed when IDL starts must have a procedure named `main` or a procedure with the same name as the `.sav` file minus the extension. The main `.sav` file is the file that is restored and run when you start your IDL application.
2. Make sure that any `.pro` files that you want to be included in your main `.sav` file have been added to your project.
3. Check blocking of widget events. Make sure your application widget events are *not* dependent upon the NO_BLOCK keyword of XMANAGER. NO_BLOCK is ignored by IDL in runtime or Virtual Machine mode. If a main procedure uses XMANAGER with the NO_BLOCK keyword set, IDL defers subsequent processing of the commands following the XMANAGER call until the widget associated with the call to XMANAGER is destroyed.

Add Required Files to Your Project

There may be files that are not part of the default IDL distribution that you need to include with your application. If you add these files to your project, they will be included in your distribution when you export your project. If you do not include these files in your project before exporting the project, you can manually add the files by copying them to your distribution:

1. If your application uses the IDL ActiveX control, IDL DataMiner, or the Network License Server for Windows NT, add the necessary files to the manifest using the instructions in "Modifying the Manifest File" on page 493.
2. If your application requires any other IDL files that are not part of the default IDL distribution, such as high-resolution maps, add them to your project or modify the manifest file. Note that only IDL files can be added to the manifest.
3. If your application requires any data files that are not in the IDL distribution, including ASCII, binary, or image files, add them to your project.

Build Your Project

You must build a project before it can be exported. For more information on the following steps, see "Setting the Options for a Project" and "Building a Project" in Chapter 20:

1. Start the IDLDE and select **File → Open Project**. Navigate to and select your project (`.prj`) file and click **Open**.

2. Select **Project → Options**. In the Project Options dialog, select the desired option in the Project Type group. The option you select depends on the type of license you have purchased and the type of application:
   - **Source File (.pro)** — If you are creating a Callable IDL or ActiveX application that makes direct calls to an IDL `.pro` file using the `IDL_Execute()` or `IDL_ExecuteStr()` functions (for Callable IDL applications) or the `ExecuteStr` method (for ActiveX applications), select this option.
   - **Save File (.sav)** — If you have purchased a runtime license, and you are creating a native IDL application, or a Callable IDL or ActiveX application that uses a `.sav` file, select this option.
   - **Licensed Save File (.sav)** — If you have purchased an embedded license, and you are creating a native IDL application, select this option. Note that this option is grayed out if you do not have an embedded license. If you have purchased an embedded license, and you are creating a Callable IDL or ActiveX application that uses a `.sav` file, it does not matter whether you choose this option or the Save File option because the licensing information is embedded in the application code rather than the `.sav` file for these types of applications.
3. In the **Run Command** field, enter the name of the IDL command that is to be called when your application is executed. By default, this is the name of your project, minus the `.prj` extension. Typically, this is the main program in your application.
4. In the **Build Command** field, specify the IDL command used to build the application. If left blank, the files in the project are built according to the Project Type specified and are compiled (if applicable) in the order specified on the Build Order tab of the project window. You can enter any valid IDL command including `.sav` or `.pro` files. You can also enter a batch file using `@filename` in order to perform other operations, such as running a Perl script on your source or data files before compiling.
5. In the **Save File** field, specify the name of the `.sav` file to create when building your project.
6. Click **OK**.
7. Select **Project → Build** from the IDLDE menu.

Exporting Your Project in Windows

See "Under Microsoft Windows" on page 490 in "Exporting a Runtime Distribution" in Chapter 20 for instructions.

Exporting Your Project in UNIX

See "Under Unix" on page 492 in "Exporting a Runtime Distribution" in Chapter 20 for instructions.

Using the make_rt Script (UNIX only)

The `make_rt` script is a UNIX command-line tool for creating an IDL distribution.Because the **Project → Export** feature is designed for creating a project and building a `.sav` file before you export an IDL distribution, it is easier to use the `make_rt` script when your application does not use a `.sav` file.

Using the `make_rt` script is essentially a three-step process:

1. Modify the `manifest_rt.txt` file. See "1. Modify the Manifest" on page 528.
2. Run `make_rt`. See "2. Run make_rt" on page 528.
   - Create destination directory.
   - Enter `make_rt` plus parameters at the UNIX command line.
3. Add any additional required files to the distribution. See "3. Add Required Files to Your Distribution" on page 528.

Before Running make_rt

Before creating the distribution, check your application for the following:

1. Verify the procedure name in your main `.sav` file. The main `.sav` file to be executed when IDL starts must have a procedure named `main` or a procedure with the same name as the `.sav` file minus the extension. The main `.sav` file is the file that is restored and run when you start your IDL application.
2. Check blocking of widget events. Make sure your application widget events are *not* dependent upon the NO_BLOCK keyword of XMANAGER. NO_BLOCK is ignored by IDL in runtime or Virtual Machine mode. If a main procedure uses XMANAGER with the NO_BLOCK keyword set, IDL defers subsequent processing of the commands following the XMANAGER call until the widget associated with the call to XMANAGER is destroyed.

1. Modify the Manifest

If your application uses IDL files that are not part of the IDL distribution, you can include these files when you run `make_rt` by modifying the manifest file. If you do not add these files to the manifest before running `make_rt`, you must manually copy the required files to your distribution.

The manifest file is located in *`idl-dir`*`/bin/manifest_rt.txt`, where *`idl-dir`* is the main IDL directory.

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

1. Open `manifest_rt.txt` in any text editor.
2. Add the path and filename of any other files in the IDL distribution that you want to include to the list of files to export. Make sure that the path is relative to the *`idl-dir`*. Note that only IDL files can be added to the manifest.
3. Make sure that you have not included any blank lines in the file.
4. Save the file.

2. Run make_rt

Run the `make_rt` script by performing the following steps:

1. Create a directory in which to locate your application distribution.
2. Open a command shell and change directories to the *`idl-dir`*`/bin` directory, where *`idl-dir`* is the main IDL directory.
3. Enter the `make_rt` command using the syntax described in "Syntax of the make_rt Script" on page 529.
4. The `make_rt` script copies IDL binaries only for the platform on which the `make_rt` script is executed. If you wish to create a distribution that supports multiple UNIX platforms, you must run the `make_rt` script on each platform you wish to support. You can specify the same destination directory each time you run the `make_rt` script, thereby creating a distribution with a `bin.`*`platform`* directory for each supported platform.

3. 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. Add the following files to your distribution. For instructions on using SAVE and RESOLVE_ALL to create `.sav` files, see "Using SAVE and RESTORE" on page 522:

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 contains an object defined in a `.pro` file, you must save the `.pro` file as a `.sav` file and manually copy the `.sav` file to your distribution tree. If the object has any inherited properties from its superclass, and the superclass is a `.pro` file, you must also include the superclass `.pro` file in your `.sav` file. Objects using a `.pro` extension typically exist in the IDL distribution's `lib` subdirectory and its subdirectories. The *IDL Reference Guide* identifies object superclasses and gives the location of the object's source code.

   For example, if you have defined an object in the file `myobject__define.pro`, and this object uses the methods of IDLgrLegend, you must save both `myobject__define.pro` and a copy of IDLgrLegend's source code, `idlgrlegend__define.pro`, in a `.sav` file and add the `.sav` file to your distribution.

   If your object files call any other routines defined in `.pro` files, you must include these `.pro` files in a `.sav` file and add them to your distribution.
3. If your application calls any routines via quoted strings, such as in CALL_PROCEDURE, CALL_FUNCTION, CALL_METHOD, EXECUTE, or in keywords that can contain procedure names such as TICKFORMAT or EVENT_PRO, you must include the `.pro` files for these routines in a `.sav` file and add them to your distribution.
4. If your application uses IDL variables, such as a custom ASCII template, or if you want to distribute other procedures and functions that are not included in your main `.sav` file, you will need to create `.sav` files using the SAVE procedure, and copy the `.sav` files to your distribution. You must save variables and procedures in separate `.sav` files. These `.sav` files can be restored by using the RESTORE procedure in your main procedure, or can be restored automatically by IDL when resolving a routine with the same name as the `.sav` file. See "Using SAVE and RESTORE" on page 522 for more information.

Syntax of the make_rt Script

The `make_rt` script uses the following syntax:

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

Following are descriptions for each parameter of the `make_rt` command:

source — The path to the main IDL directory, such as `/usr/local/rsi/idl_6.0`. If not specified, you will be prompted.

dest — The full path to the destination directory that will contain the distribution. This directory must already exist—`make_rt` will not create this directory.

manifest — The full path and filename containing the manifest file. This is the `manifest_rt.txt` file in the *`idl-dir`*`\bin` directory.

savefile — The name of the executable used to start your application, without any extension. For IDL applications, this is typically the name of your `.sav` file. For Callable IDL applications, the IDL executable, located in the `bin.`*`platform`* directory, is always named `idl`. The value specified for the `savefile` parameter determines the name of your application startup script, located in the top-level directory. For Callable IDL applications, this script must be edited.

mode — The type of `.sav` file your application uses. Valid values are:

- rt — Use this value if your application uses a `.sav` file that does not contain an embedded license (i.e., you have purchased a runtime license). This value is the equivalent of selecting the **Save File (.sav)** option in the **Project** → **Options** dialog.
- em — Use this value if you have a licensed `.sav` file (.i.e., you have purchased an embedded license). This value is the equivalent of selecting the **Licensed Save File (.sav)** option in the **Project** → **Options** dialog.

If your application does not use a `.sav` file, it does not matter which value you use, but you must specify a value.

Example: Callable IDL Application

Assume you have IDL installed in the `/usr/local/rsi/idl_6.0` directory, and you are creating a distribution in the /`myapps` directory. To create a distribution for a Callable IDL application that does not use a `.sav` file, you would execute the following command from the `/usr/local/rsi/idl_6.0/bin` directory:

```
make_rt /myapps /usr/local/rsi/idl_6.0/bin/manifest_rt.txt myapp rt
```

Note

The `savefile` and `mode` parameters are required, even if your application does not use a `.sav` file. For Callable IDL applications, the `savefile` parameter specifies the name of your application startup script in the top-level directory of your application distribution.

Preparing a Distribution

The method for preparing an IDL distribution varies depending on the platform you are using.

Note

Applications that run in the IDL Virtual Machine or in an end-user's existing IDL installation do not need an IDL distribution.

Windows

The following sections describe the steps that must be performed prior to distributing your application on Windows.

Preparing IDL Applications or Callable IDL Applications

1. Edit the idl.ini File

IDL uses an initialization file called `idl.ini` to supply software licensing information and program defaults. IDL reads the `idl.ini` file when it starts up for the first time and automatically converts the file entries into Windows Registry entries. Even if you choose to have your customer license IDL the first time your application is run, you must supply an `idl.ini` file. The `idl.ini` file resides in the `bin\bin.x86` directory.

The `idl.ini` file is automatically created for you when you create an IDL distribution. You might need to make some changes to this file to ensure that your application starts correctly on the end user's system. The following is an example of the default `idl.ini` file that will be copied to the `bin\bin.x86` directory of your distribution:

```
[IDL 6.0]
HomeDir=..\..
PortSetId=0
RuntimeFile=..\..\myapp.sav
SearchPath=+..\..
```

Note

All directories specified in the `idl.ini` file should be relative to the directory containing the `.ini` file. This ensures that if the directory tree for your application is moved to another location, it will still run.

The general syntax of the `.ini` file is:

Field=Value

There are additional fields available that are not included in the default `.ini` file. The available fields in the `.ini` file are as follows:

- **HomeDir** — Specifies the directory that contains the `bin` directory.
- **HelpPath** — Specifies the directories in which IDL will search for help files. The "+" symbol at the beginning of the string indicates that all subdirectories of the specified directory should be searched. Separate multiple directories with a semicolon. For example:

```
HelpPath=..\..;..\helpfiles
```

- **RSI Root** — Specifies the directory that contains the `license` subdirectory for use with software-based node-locked licenses.This field is not included by default.
- **RuntimeFile** — Specifies the name of the `.sav` file to be automatically restored when IDL starts. If this field is left blank and no `.sav` file is specified at the command line, IDL will attempt to restore a file named `runtime.sav` in the directory specified by the `HomeDir` field.
- **RuntimeIcon** — Specifies the path and name of the application custom icon (`.ico` file).
- **SearchPath** — Specifies the directories in which IDL will search for `.sav` files when attempting to resolve routines. The "+" symbol at the beginning of the string indicates that all subdirectories of the specified directory should be searched. Separate multiple directories with a semicolon. For example:

```
SearchPath=..\..;..\savefiles
```

2. Starting Your Windows Application

You must provide your end user with instructions on starting your application. You can simply provide the end user with the name and location of your application executable, or if you are using an installer for your application, you can create shortcuts and/or Start menu items for your user.

In your application distribution, there is a `bin\bin.x86` directory containing the following:

- The executable for your application. For IDL applications, this is an executable with the name specified in the Run Command field of the **Project → Options** dialog (if you used the **Project → Export** feature), or the `savefile` parameter (if you used the `make_rt` script). For example, if your `.sav` file is called `myapp.sav`, an executable with the name `myapp.exe` will be created.

For Callable IDL or ActiveX applications, this is the executable you have created.

- The IDL executable. For IDL applications, the IDL executable is the same file as your application executable as described above. For Callable IDL or ActiveX applications, the IDL executable is a separate file from your application executable, and can be deleted.
- The `idl.ini` file. This file tells IDL where to find `.sav` files (if your application uses any), and where to find the license file (for applications that do not contain an embedded license).

You can start your application by simply double-clicking your application `.exe` file in the `bin\bin.x86` directory. If your application uses a `.sav` file, this executable does the following:

- Restores the specified `.sav` file, if one is specified at the command line.
- Restores the `.sav` file specified in the `idl.ini` file if no `.sav` file is specified at the command line.
- Restores the file `runtime.sav`, if no file is specified at the command line or in the `idl.ini` file.

IDL then calls the main procedure. This is:

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

When the main procedure returns, IDL exits.

If you wish to create a shortcut or Start menu item for your user, you would use the path to your application executable. For example, assume the top-level directory of your application is `C:\myapp`, which contains the file `myapp.sav`, the executable `myapp.exe` is located in `C:\myapp\bin\bin.x86` directory, and your `idl.ini` file looks like this:

```
[IDL 6.0]
HomeDir=..\..
PortSetId=0
LicenseMethod=1
RuntimeFile=..\..\myapp.sav
SearchPath=+..\..
RSI Root=..\..\..
```

To create a shortcut that restores the file `myapp.sav`, you could use the following for the Command Line field of your shortcut:

```
C:\myapp\bin\bin.x86\myapp.exe
```

If your .sav file name is different from the name of your executable, you can specify the name of the .sav file to be restored. For example, if your executable is named myapp.exe, and your .sav file is named mysav.sav, you could use the following for the Command Line field of your shortcut:

```
C:\myapp\bin\bin.x86\myapp.exe C:\myapp\mysav.sav
```

In order for this to work, your .sav file must contain a procedure named main.

3. Rename idl.000 and Remove reg.dat

If you have launched your application prior to distributing the application, then the idl.ini file has been renamed idl.000 and a reg.dat file has been created. Before you distribute your application, rename idl.000 back to idl.ini and remove the reg.dat file from the bin\bin.x86 directory. Since reg.dat is a hidden file, you may need to select **View → Options → Show all files** in Windows Explorer and click **OK** to see it.

Preparing ActiveX Applications

When an ActiveX application is installed on the end user's system, the following steps must be performed. These tasks can be performed using an installer, or they can be performed manually by the end user:

1. Make sure that when your application is being installed, the idldrawx3.ocx file from the bin.x86 directory of your distribution tree is transferred to the winnt\system32 directory.
2. Register the idldrawx3.ocx file during your application installation process. This can be accomplished using the regsvr32.exe executable. For example, your installation script could contain the following command:

   ```
   regsvr32 idldrawx3.ocx
   ```

   For more information, refer to your Microsoft Windows documentation.

UNIX

This section describes the steps that are required before distributing your application on UNIX, as well as the steps required to start your application.

IDL Applications

No special preparation for distribution is required.

Callable IDL Applications

The following steps are required before distributing a Callable IDL application.

1. Add Your Executables to the Distribution

Copy your application executable 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

Located in the `bin` directory of your distribution is a script called `idl`. 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 `savefile` parameter you specified when you ran the `make_rt` script. Make the following changes to this script:

1. For Callable IDL applications, you must edit this startup script to execute the script in the `bin` directory that you renamed in the previous section, "Rename the idl Script". 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.

2. 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 RSI 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
   ```

Starting Your UNIX Application

For both IDL and Callable IDL applications, starting your application on UNIX requires the following steps:

1. Set the IDL_DIR environment variable as described above.
2. Execute the startup script in the top-level directory of your application.

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.

Your users should keep the following points in mind when installing your application:

- In order to avoid any possible conflicts with existing versions of IDL, do NOT install your application in the same directory as IDL x.x, where IDL x.x is the version used by your application.
- If prompted with a dialog asking whether or not to import initialization preferences, answer **Yes**. Answering **No** may mean that you are not licensing IDL.

Note

RSI's Global Services group can create installation packages for your application. Contact your RSI sales representative for additional information.

Incorporating the IDL Data Miner

If your application uses IDL DataMiner, you will 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

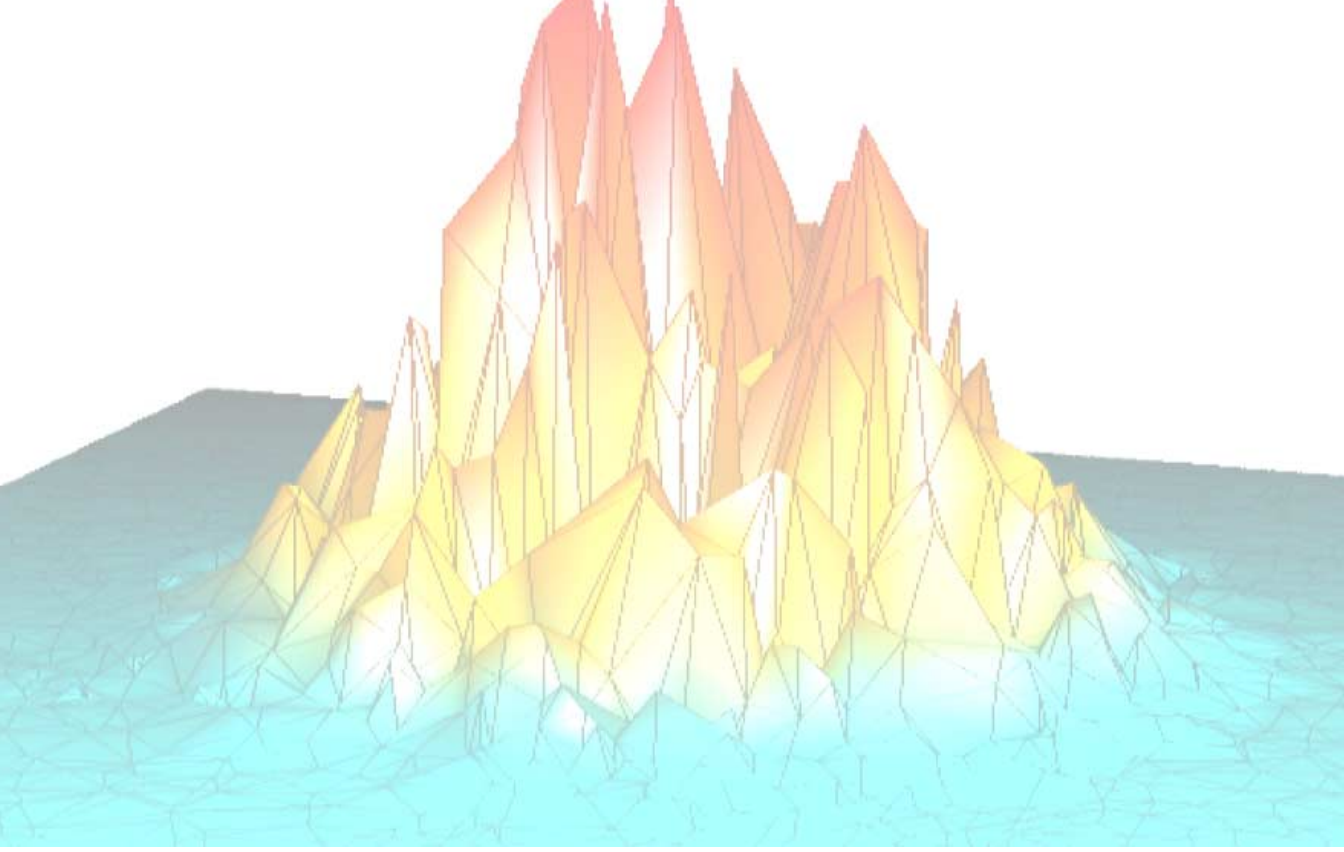
If your application uses IDL DataMiner, please call RSI Technical Support for instructions.

- E-mail: `support@RSInc.com`
- Phone: (303) 413-3920

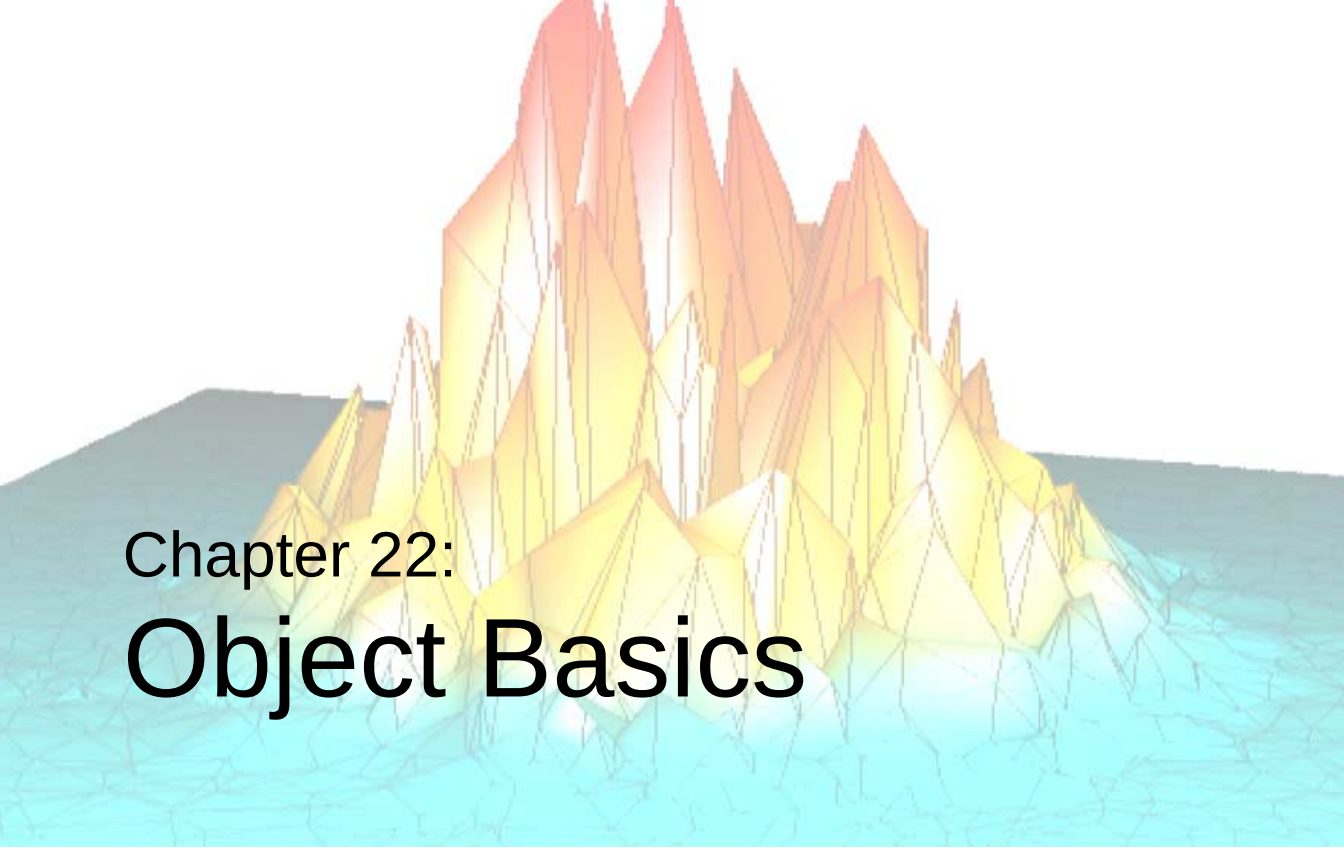
UNIX

If your application uses IDL DataMiner, you will need to add the files listed in the `manifest_aux.txt` file to the manifest file before creating your IDL distribution.

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.



Part IV: Using IDL Objects



Chapter 22:
Object Basics

The following topics are covered in this chapter:

Object-Oriented Programming

Traditional programming techniques make a strong distinction between routines written in the programming language (procedures and functions in the case of IDL) and data to be acted upon by the routines. *Object oriented* programming begins to remove this distinction by melding the two into *objects* that can contain both routines and data. Object orientation provides a layer of abstraction that allows the programmer to build robust applications from groups of reusable elements.

Beginning in version 5.0, IDL provides a set of tools for developing object-oriented applications. IDL's Object Graphics engine is object-oriented, and a class library of graphics objects allows you to create applications that provide equivalent graphics functionality regardless of your (or your users') computer platform, output devices, etc. As an IDL programmer, you can use IDL's traditional procedures and functions as well as the new object features to create your own object modules. Applications built from object modules are, in general, easier to maintain and extend than their traditional counterparts.

This chapter describes how to use object techniques with IDL. A complete discussion of object orientation is beyond the scope of this book—if you are new to object oriented programming, consult one of the many references on object oriented program that are available.

IDL Object Overview

IDL objects are actually special *heap variables*, which means that they are global in scope and provide explicit user control over their lifetimes. Object heap variables can only be accessed via object references. Object references are discussed in this chapter. Heap variables in general are discussed in detail in “Heap Variables” on page 167.

Briefly, IDL provides support for the following object concepts and mechanisms:

Classes and Instances

IDL objects are created as *instances* of a *class*, which is defined in the form of an IDL structure. The name of the structure is also the class name for the object. The *instance data* of an object is an IDL structure contained in the object heap variable, and can only be accessed by special functions and procedures, called *methods*, which are associated with the class. Class structures are discussed in “Class Structures” on page 547.

Encapsulation

Encapsulation is the ability to combine data and the routines that affect the data into a single object. IDL accomplishes this by only allowing access to an object’s instance data via that object’s *methods*. Data contained in an object is hidden from all but the object’s own methods.

Methods

IDL allows you to define method procedures and functions using all of the programming tools available in IDL. Method routines are identified as belonging to an object class via a routine naming convention. Methods are discussed in detail in “Method Routines” on page 562.

Polymorphism

Polymorphism is the ability to create multiple object types that support the same operations. For example, many of IDL’s graphics objects support an operation called “Draw,” which sends graphics output to a specified place. The “Draw” operation is different in different contexts; sending a graphic to a printer is different from writing it to a file. Polymorphism allows the details of the differences to remain hidden—all you need to know is that a given object supports the “Draw” operation.

Inheritance

Inheritance is the ability of an object class to inherit the behavior of other object classes. This means that when writing a new object class that is very much like an existing object class, you need only program the functions that are different from those in the inherited class. IDL supports multiple inheritance—that is, an object can inherit qualities from any number of other existing object classes. Inheritance is discussed in detail in “Inheritance” on page 549.

Persistence

Persistence is the ability of objects to remain in existence in memory after they have been created, allowing you to alter their behavior or appearance after their creation. IDL objects persist until you explicitly destroy them, or until the end of the IDL session. In practice, object persistence removes the need (in traditional IDL programs) to re-execute IDL commands that create an item (a plot, for example) in order to change a detail of the item. For example, once you have created a graphic object containing a plot, you can alter any aspect of the plot “on the fly,” without re-creating it. Similarly, having created an object containing a plot, you need not recreate the plot in order to print, save to an image file, or re-display it.

IDL objects also persist in the sense that you can use the SAVE and RESTORE routines to save and recreate objects between IDL sessions.

Class Structures

Object instance data is contained in named IDL structures. We will use the term *class structure* to refer to IDL structures containing object instance data.

Beyond the restriction that class structures must be named structures, there are no limits on what a class structure contains. Class structures can include data of any type or organization, including pointers and object references. When an object is created, the name of the class structure becomes the name of the class itself, and thus serves to define the names of all methods associated with the class. For example, if we create the following class structure:

```
struct = { Class1, data1:0L, data2:FLTARR(10) }
```

any objects created from the class structure Class1 would have the same two fields (data1, a long integer, and data2, a ten-element floating-point array) and any methods associated with the class would have the name Class1::*method*, where *method* is the actual name of the method routine. Methods are discussed in detail in "Method Routines" on page 562.

Note

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

It is important to realize that creating a class structure does not create an object. Objects can only be created by calling the OBJ_NEW or OBJARR function with the name of the class structure as the argument, and can only be accessed via the returned object reference. In addition, object methods can only be called on objects, and not on class structures themselves.

Once defined, a given class structure type cannot be changed. If a structure definition is executed and the structure already exists, each tag name and the structure of each tag field must agree with the original definition. To redefine a structure, you must either reset or exit the current IDL session.

Automatic Class Structure Definition

If IDL finds a reference to a structure that has not been defined, it will search for a structure definition procedure to define it. (This is true of all structure references, not just class structures.) Automatic structure definition is discussed in “Automatic Structure Definition” on page 159. Briefly, if IDL encounters a structure reference for a structure type that has not been defined, it searches for a routine with a name of the form

```
STRUCT__DEFINE
```

where STRUCT is the name of the structure type. Note that there are *two* underscores in the name of the structure definition routine.

The following is an example of a structure definition procedure that defines a structure that will be used for the class CNAME.

```
PRO CNAME__DEFINE
    struct = { CNAME, data1:0L, data2:FLTARR(10) }
END
```

This defines a structure named CNAME with 2 data fields (data1, a long integer, and data2, a ten-element floating-point array). If you tell IDL to create an object of type CNAME before this structure has been defined, IDL will search for the procedure CNAME__DEFINE to define the class structure before attempting to create the object. If the CNAME__DEFINE procedure has not yet been compiled, IDL will use its normal routine searching algorithm to attempt to find a file named CNAME__DEFINE.PRO. If IDL cannot find a defined structure or structure definition routine, the object-creation operation will fail.

Note

If you are creating structure definitions on the fly, the possibility exists that you will run into namespace conflicts — that is, a structure with the same name as the structure you are attempting to create may already exist. This can be a problem if you are developing object-oriented applications for others, since you probably do not have much control over the IDL environment on your clients’ systems. You can avoid most problems by creating a unique namespace for your routines; RSI does this by prefixing the names of objects with the letters “IDL”. To be completely sure that the objects created by your programs are what you expect, however, you should have the program inspect the created structures and handle errors appropriately.

Inheritance

When defining a class structure, use the INHERITS specifier to indicate that this structure inherits instance data and methods from another class structure. For example, if we defined a class structure called “circle,” as follows:

```
struct = { circle, x:0, y:0, radius:0 }
```

we can define a subclass of the “circle” class like this:

```
struct = { filled_circle, color:0, INHERITS circle }
```

You can use the INHERITS specifier in any structure definition. However, when the structure being defined is a *class structure* (that is, an object will be created from the structure), inheritance affects both the structure definition and the object methods available to the object that inherits. The INHERITS specifier is discussed in “Structure Inheritance” on page 146.

When a class structure inherits from another class structure, it is said to be a *subclass* of the class it inherits from. Similarly, the class that is inherited from is called a *superclass* of the new class. Defining a subclass of an existing class in this manner has two consequences. First, the class structure for the subclass is constructed as if the elements of the inherited class structure were included in-line in the structure definition. In our example, the command defining the “filled_circle” class above would create the followings structure definition:

```
{ filled_circle, color:0, x:0, y:0, radius:0 }
```

Note that the data fields from the inherited structure definition appear in-line at the point where the INHERITS specifier appears.

The second consequence of defining a subclass structure that inherits from another class structure is that when an object is created from the subclass structure, that object inherits the *methods* of the superclass as well as its data fields. That is, if an object of the superclass type has a method, that method is available to objects created from the subclass as well. In our example above, say we create an object of type circle and define a Print method for it. Any objects of type filled_circle will also have access to the Print method defined for circle.

IDL allows multiple inheritance. This means that you can include the INHERITS specifier as many times as you desire in a structure definition, *as long as all of the resulting data fields have unique names*. Data fields must have unique names because when the class structure definition is built, the tag names are included in-line at the point where the INHERITS specifier appears. Duplicate tag names will cause the structure definition to fail; it is your responsibility as a programmer to ensure that tag names are not used more than once in a structure definition.

Note

The requirement that names be unique applies only to *data* fields. It is perfectly legitimate (and often necessary) for subclasses to have methods with the same names as methods belonging to the superclass. See "Method Overriding" on page 566 for details.

If a structure referred to by an INHERITS specifier has not been defined in the current IDL session, IDL will attempt to define it in the manner described in "Automatic Class Structure Definition" on page 548.

Object Heap Variables

Object heap variables are IDL heap variables that are accessible only via *object references.* While there are many similarities between object references and pointers, it is important to understand that they are not the same type, and cannot be used interchangeably. Object heap variables are created using the OBJ_NEW and OBJARR functions. For more information on heap variables and pointers, see "IDL Pointers" on page 172.

Heap variables are a special class of IDL variables that have global scope and explicit user control over their lifetime. They can be basic IDL variables, accessible via pointers, or objects, accessible via object references. In IDL documentation of pointers and objects, heap variables accessible via pointers are called *pointer heap variables*, and heap variables accessible via object references are called *object heap variables*.

Note

Pointers and object references have many similarities, the strongest of which is that both point at heap variables. It is important to understand that they are not the same type, and cannot be used interchangeably. Pointers and object references are used to solve different sorts of problems. Pointers are useful for building dynamic data structures, and for passing large data around using a lightweight token (the pointer itself) instead of copying data. Objects are used to apply object oriented design techniques and organization to a system. It is, of course, often useful to use both in a given program.

Heap variables are global in scope, but do not suffer from the limitations of COMMON blocks. That is, heap variables are available to all program units at all times. (Remember, however, that IDL variables containing pointers to heap variables are *not* global in scope and must be declared in a COMMON block if you want to share them between program units.)

Heap variables:

- Facilitate object oriented programming.
- Provide full support for Save and Restore. Saving a pointer or object reference automatically causes the associated heap variable to be saved as well. This means that if the heap variable contains a pointer or object reference, the heap variables they point to are also saved. Complicated self-referential data structures can be saved and restored easily.

- Are manipulated primarily via pointers or object references using built in language operators rather than special functions and procedures.
- Can be used to construct arbitrary, fully general data structures in conjunction with pointers.

Dangling References

If a heap variable is destroyed, any remaining pointer variable or object reference that still refers to it is said to contain a *dangling reference*. Unlike lower level languages such as C, dereferencing a dangling reference will not crash or corrupt your IDL session. It will, however, fail with an error message.

There are several possible approaches to avoiding such errors. The best option is to structure your code such that dangling references do not occur. You can, however, verify the validity of pointers or object references before using them (via the PTR_VALID or OBJ_VALID functions) or use the CATCH mechanism to recover from the effect of such a dereferencing.

Heap Variable “Leakage”

Heap variables are not reference counted—that is, IDL does not keep track of how many references to a heap variable exist, or stop the last such reference from being destroyed—so it is possible to lose access to them and the memory they are using. See “Heap Variables” on page 167 for additional details.

Freeing Heap Variables

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" in the *IDL Reference Guide* manual for further details.

Null Objects

The *Null Object* is a special object reference that is guaranteed to never point at a valid object heap variable. It is used by IDL to initialize object reference variables when no other initializing value is present. It is also a convenient value to use when defining structure definitions for fields that are object references, since it avoids the need to have a pre-existing valid object reference.

Null objects are created when you call an object-creation routine but do not specify a class structure to be used as the new object's template. The following statement creates a null object:

```
nullobj = OBJ_NEW()
```

The Object Lifecycle

As discussed above, objects are *persistent*, meaning they exist in memory until you destroy them. We can break the life of an object into three phases: creation and initialization, use, and destruction. Object *lifecycle routines* allow the creation and destruction of object references; *lifecycle methods* associated with an object allow you to control what happens when an object is created or destroyed.

This section will discuss the first and last phases of the object lifecycle; the remainder of this chapter discusses manipulation of existing objects and use of object method routines.

Creation and Initialization

Object references are created using one of two lifecycle routines: OBJ_NEW or OBJARR. Newly created objects are initialized upon creation in two ways:

1. The object reference is created based on the class structure specified,
2. The object's INIT method (if it has one) is called to initialize the object's instance data (contained in fields defined by the class structure). If the object does not have an INIT method, the object's superclasses (if any) are searched for an INIT method.

The INIT Method

An object's lifecycle method INIT is a function named *Class*::INIT (where *Class* is the actual name of the class). The purpose of the INIT method is to populate a newly-created object with instance data. INIT should return a scalar TRUE value (such as 1) if the initialization is successful, and FALSE (such as 0) if the initialization fails.

The INIT method is unusual in that it *cannot be called outside an object-creation operation*. This means that—unlike most object methods—you cannot call the INIT method on an object directly. You can, however, call an object's INIT method from within the INIT method of a subclass of that object. This allows you to specify parameters used by the superclass' INIT method along with those used by the INIT method of the object being created. In practice, this is often done using the _EXTRA keyword. See"Keyword Inheritance" on page 79 for details.

The OBJ_NEW Function

Use the OBJ_NEW function to create an object reference to a new object heap variable. If you supply the name of a class structure as its argument, OBJ_NEW creates a new object containing an instance of that class structure. Note that the fields

of the newly-created object's instance data structure will all be empty. For example, the command:

```
obj1 = OBJ_NEW('ClassName')
```

creates a new object heap variable that contains an instance of the class structure *ClassName*, and places an object reference to this heap variable in obj1. If you do not supply an argument, the newly-created object will be a null object.

When creating an object from a class structure, OBJ_NEW goes through the following steps:

1. If the class structure has not been defined, IDL will attempt to find and call a procedure to define it automatically. See "Automatic Class Structure Definition" on page 548 for details. If the structure is still not defined, OBJ_NEW fails and issues an error.
2. If the class structure has been defined, OBJ_NEW creates an object heap variable containing a zeroed instance of the class structure.
3. Once the new object heap variable has been created, OBJ_NEW looks for a *method* function named *Class*::INIT (where *Class* is the actual name of the class). If an INIT method exists, it is called with the new object as its implicit SELF argument, as well as any arguments and keywords specified in the call to OBJ_NEW. If the class has no INIT method, the usual method-searching rules are applied to find one from a superclass. For more information on methods and method-searching rules, see "Method Routines" on page 562.

Note

OBJ_NEW does not call all the INIT methods in an object's class hierarchy. Instead, it simply calls the first one it finds. Therefore, the INIT method for a class should call the INIT methods of its direct superclasses as necessary.

4. If the INIT method returns true, or if no INIT method exists, OBJ_NEW returns an object reference to the heap variable. If INIT returns false, OBJ_NEW destroys the new object and returns the NULL object reference, indicating that the operation failed. Note that in this case the CLEANUP method is not called.

See "OBJ_NEW" in the *IDL Reference Guide* manual for further details.

The OBJARR Function

Use the OBJARR function to create an array of objects of up to eight dimensions. Every element of the array created by OBJARR is set to the null object. For example,

the following command creates a 3 by 3 element object reference array with each element contain the null object reference:

```
obj2 = OBJARR(3, 3)
```

See "OBJARR" in the *IDL Reference Guide* manual for further details.

Destruction

Use the OBJ_DESTROY procedure to destroy an object. If the object's class, or one of its superclasses, supplies a procedure method named CLEANUP, that method is called, and all arguments and keywords passed by the user are passed to it. The CLEANUP method should perform any required cleanup on the object and return. Whether a CLEANUP method actually exists or not, IDL will destroy the heap variable representing the object and return.

The CLEANUP method is unusual in that it *cannot be called outside an object-destruction operation*. This means that—unlike most object methods—you cannot call the CLEANUP method on an object directly. You can, however, call an object's CLEANUP method from within the CLEANUP method of a subclass of that object.

Note that the object references themselves are not destroyed. Object references that refer to nonexistent object heap variables are known as dangling references, and are discussed in more detail in "Dangling References" on page 179.

See "OBJ_DESTROY" in the *IDL Reference Guide* manual for further details.

Operations on Objects

Object reference variables are not directly usable by many of the operators, functions, or procedures provided by IDL. You cannot, for example, do arithmetic on them or plot them. You can, of course, do these things with the contents of the structures contained in the object heap variables referred to by object references, assuming that they contain non-object data.

There are four IDL operators that work with object reference variables: assignment, method invocation, EQ, and NE. In addition, the structure dot operator (.) is allowed *within methods of a class*. The remaining operators (addition, subtraction, etc.) do not make any sense for object references and are not defined.

Many non-computational functions and procedures in IDL do work with object references. Examples are SIZE, N_ELEMENTS, HELP, and PRINT. It is worth noting that the only I/O allowed directly on object reference variables is default formatted output, in which they are printed as a symbolic description of the heap variable they refer to. This is merely a debugging aid for the IDL programmer—input/output of object reference variables does not make sense in general and is not allowed. Please note that this does *not* imply that I/O on the contents of non-object instance data contained in heap variables is not allowed. Passing non-object instance data contained in an object heap variable to the PRINT command is a simple example of this type of I/O.

Assignment

Assignment works in the expected manner—assigning an object reference to a variable gives you another variable with the same reference. Hence, after executing the statements:

```
;Define a class structure.
struct = { cname, data1:0.0 }

;Create an object.
A = OBJ_NEW('cname')

;Create a second object reference.
B = A

HELP, A, B
```

IDL prints:

```
A               OBJREF    = <ObjHeapVar1(CNAME)>
B               OBJREF    = <ObjHeapVar1(CNAME)>
```

Note that both A and B are references to the same object heap variable.

Method Invocation

In order to perform an action on an object's instance data, you must call one of the object's *methods*. (See "Method Routines" on page 562 for more on methods.) To call a method, you must use the method invocation operator, -> (the hyphen followed by the greater-than sign). The syntax is:

ObjRef -> Method

where *ObjRef* is an object reference and *Method* is a method belonging either to the object's class or to one of its superclasses. *Method* may be specified either partially (using only the method name) or completely using both the class name and method name, connected with two colons:

ObjRef -> Class::Method

Equality and Inequality

The EQ and NE operators allow you to compare object references to see if they refer to the same object heap variable. For example:

```
;Define a class structure.
struct = {cname, data:0.0}

;Create an object.
A = OBJ_NEW('CNAME')

;B refers to the same object as A.
B = A

;C contains a null object reference.
C = OBJ_NEW()

PRINT, 'A EQ B: ', A EQ B & $
PRINT, 'A NE B: ', A NE B & $
PRINT, 'A EQ C: ', A EQ C & $
PRINT, 'C EQ NULL: ', C EQ OBJ_NEW() & $
PRINT, 'C NE NULL:', C NE OBJ_NEW()
```

IDL prints:

```
A EQ B:    1
A NE B:    0
A EQ C:    0
C EQ NULL: 1
C NE NULL: 0
```

Obtaining Information about Objects

Three IDL routines allow you to obtain information about an existing object:

OBJ_CLASS

Use the OBJ_CLASS function to obtain the class name of a specified object, or to obtain the names of a specified object's direct superclasses. For example, if we create the following class structures:

```
struct = {class1, data1:0.0 }
struct = {class2, data2a:0, data2b:0L, INHERITS class1 }
```

We can now create an object and use OBJ_CLASS to determine its class and superclass membership.

```
;Create an object.
A = OBJ_NEW('class2')

;Print A's class membership.
PRINT, OBJ_CLASS(A)
```

IDL prints:

```
CLASS2
```

Or you can print as superclasses:

```
;Print A's superclasses.
PRINT, OBJ_CLASS(A, /SUPERCLASS)
```

IDL prints:

```
CLASS1
```

See "OBJ_CLASS" in the *IDL Reference Guide* manual for further details.

OBJ_ISA

Use the OBJ_ISA function to determine whether a specified object is an instance or subclass of a specified object. For example, if we have defined the object A as above:

```
IF OBJ_ISA(A, 'class2') THEN $
    PRINT, 'A is an instance of class2.'
```

IDL prints:

```
A is an instance of class2.
```

See "OBJ_ISA" in the *IDL Reference Guide* manual for further details.

OBJ_VALID

Use the OBJ_VALID function to verify that one or more object references refer to valid and currently existing object heap variables. If supplied with a single object reference as its argument, OBJ_VALID returns TRUE (1) if the reference refers to a valid object heap variable, or FALSE (0) otherwise. If supplied with an array of object references, OBJ_VALID returns an array of TRUE and FALSE values corresponding to the input array. For example:

```
;Create a class structure.
struct = {cname, data:0.0}

;Create a new object.
A = OBJ_NEW('CNAME')

IF OBJ_VALID(A) PRINT, "A refers to a valid object." $
    ELSE PRINT, "A does not refer to a valid object."
```

IDL prints:

```
A refers to a valid object.
```

If we destroy the object:

```
;Destroy the object.
OBJ_DESTROY, A

IF OBJ_VALID(A) PRINT, "A refers to a valid object." $
    ELSE PRINT, "A does not refer to a valid object."
```

IDL prints:

```
A does not refer to a valid object.
```

See “OBJ_VALID” in the *IDL Reference Guide* manual for further details.

Method Routines

IDL objects can have associated procedures and functions called *methods*. Methods are called on objects via their object references using the method invocation operator.

While object methods are constructed in the same was as any other IDL procedure or function, they are different from other routines in the following ways:

- Object methods are defined using a special naming convention that incorporates the name of the class to which the method belongs.
- All method routines automatically pass an implicit argument named self, which contains the object reference of the object on which the method is called.
- Object methods cannot be called on their own. You must use the method invocation operator and supply a valid object reference, either of the class the method belongs to or of one of that class' subclasses.

Defining Method Routines

Method routines are defined in the same way as other IDL procedures and functions, with the exception that the name of the class to which they belong, along with two colons, is prepended to the method name:

```
PRO ClassName::Method
    IDL statements
END
```

or

```
FUNCTION ClassName::Method, Argument1
    IDL statements
RETURN, value
END
```

For example, suppose we create two objects, each with its own "print" method.

First, define two class structures:

```
struct = { class1, data1:0.0 }
struct = { class2, data2a:0, data2b:0L, INHERITS class1 }
```

Now we define two "print" methods to print the contents of any objects of either of these two classes. (If you are typing this at the IDL command line, enter the .RUN command before each of the following procedure definitions.)

```
PRO class1::Print1
    PRINT, self.data1
```

```
END
PRO class2::Print2
    PRINT, self.data1
    PRINT, self.data2a, self.data2b
END
```

Once these procedures are defined, any objects of class1 have access to the method Print1, and any objects of class2 have access to both Print1 and Print2 (because class2 is a subclass of—it *inherits* from—class1). Note that the Print2 method prints the data1 field inherited from class1.

Note

It is not necessary to give different method names to methods from different classes, as we have done here with Print1 and Print2. In fact, in most cases both methods would have simply been called Print, with each object class knowing only about its own version of the method. We have given the two procedures different names here for instructional reasons; see "Method Overriding" on page 566 for a more complete discussion of method naming.

The Implicit *Self* Argument

Every method routine has an implicit argument parameter named self. The self parameter always contains the object reference of the object on which the method is called. In the method routines created above, self is used to specify which object the data fields should be printed from.

You do not need to explicitly pass the self argument; in fact, if you try to specify an argument called self when defining a method routine, IDL will issue an error.

Calling Method Routines

You must use the method invocation operator (->) to call a method on an object. The syntax is slightly different from other routine invocations:

```
;For a procedure method.
ObjRef -> Method

;For a function method.
Result = ObjRef -> Method()
```

Where *ObjRef* is an object reference belonging to the same class as the *Method*, or to one of that class' subclasses. We can illustrate this behavior using the Print1 and Print2 methods defined above.

First, define two new objects:

```
A = OBJ_NEW('class1')
B = OBJ_NEW('class2')
```

We can call Print1 on the object A as follows:

```
A -> Print1
```

IDL prints:

```
0.00000
```

Similarly, we can call Print2 on the object B:

```
B -> Print2
```

IDL prints:

```
0.00000
0           0
```

Since the object B inherits its properties from class1, we can also call Print1 on the object B:

```
B -> Print1
```

IDL prints:

```
0.00000
```

We cannot, however, call Print2 on the object A, since class1 does not inherit the properties of class2:

```
A -> Print2
```

IDL prints:

```
% Attempt to call undefined method: 'CLASS1::PRINT2'.
```

Searching for Method Routines

When a method is called on an object reference, IDL searches for it as with any procedure or function, and calls it if it can be found, following the naming convention established for structure definition routines. (See “Automatic Class Structure Definition” on page 548.) In other words, IDL discovers methods as it needs them in the same way as regular procedures and functions, with the exception that it searches for files named

classname__method.pro

rather than simply

method.pro

Remember that there are two *underscores* in the file name, and two *colons* in the method routine's name.

Note

If you are working in an environment where the length of filenames is limited, you may want to consider defining all object methods in the same .pro file you use to define the class structure. This practice avoids any problems caused by the need to prepend the *classname* and the two underscore characters to the method name. If you must use different .pro files, make sure that all class (and superclass) definition filenames are unique in the first eight characters.

Method Overriding

Unlike data fields, method names can be duplicated. This is an important feature that allows method overriding, which in turn facilitates polymorphism in the design of object-oriented programs. Method overriding allows a subclass to provide its own implementation of a method already provided by one of its superclasses. When a method is called on an object, IDL searches for a method of that class with that name. If found, the method is called. If not, the methods of any inherited object classes are examined in the order their INHERITS specifiers appear in the structure definition, and the first method found with the correct name is called. If no method of the specified name is found, an error occurs.

The method search proceeds *depth first, left to right*. This means that if an object's class does not provide the method called directly, IDL searches through inherited classes by first searching the leftmost included class—and all of its superclasses—before proceeding to the next inherited class to the right. If a method is defined by more than a single inherited structure definition, the first one found is used and no warning is generated. This means that class designers should pick non-generic names for their methods as well as their data fields. For example, suppose we have defined the following classes:

```
struct = { class1, data1}
struct = { class2, data2a:0, data2b:0.0, inherits class1 }
struct = { class3, data3:'', inherits class2, inherits class1 }
struct = { class 4, data4:0L, inherits class2, inherits class3 }
```

Furthermore, suppose that both class1 and class3 have a method called Print defined.

Now suppose that we create an object of class4, and call the Print method:

```
A = OBJ_NEW('class4')
A -> Print
```

IDL takes the following steps:

1. Searches class4 for a Print method. It does not find one.
2. Searches the leftmost inherited class (class2) in the class definition structure for a Print method. It does not find one.
3. Searches any superclasses of class2 for a Print method. It finds the class1 Print method and calls it on A.

Notice that IDL stops searching when it finds a method with the proper name. Thus, IDL doesn't find the Print method that belongs to class3.

Specifying Class Names in Method Calls

If you specify a class name when calling an object method, like so:

```
ObjRef -> classname::method
```

Where *classname* is the name of one of the object's superclasses, IDL will search *classname* and any of *classname's* superclasses for the method name. IDL will *not* search the object's own class or any other classes the object inherits from.

This type of method call is especially useful when a class has a method that overrides a superclass method and does its job by calling the superclass method and then adding functionality. In our simple example from "Calling Method Routines" on page 563, above, we could have defined a Print method for each class, as follows:

```
PRO class1::Print
    PRINT, self.data1
END
PRO class2::Print
    self -> class1::Print
    PRINT, self.data2a, self.data2b
END
```

In this case, to duplicate the behavior of the Print1 and Print2 methods, we make the following method calls:

```
A -> Print
```

IDL prints:

```
0.00000
```

And now the B:

```
B -> Print
```

IDL prints:

```
0.00000
0           0
```

Now we'll use the second method:

```
B -> class1::Print
```

IDL prints:

```
0.00000
```

And now A:

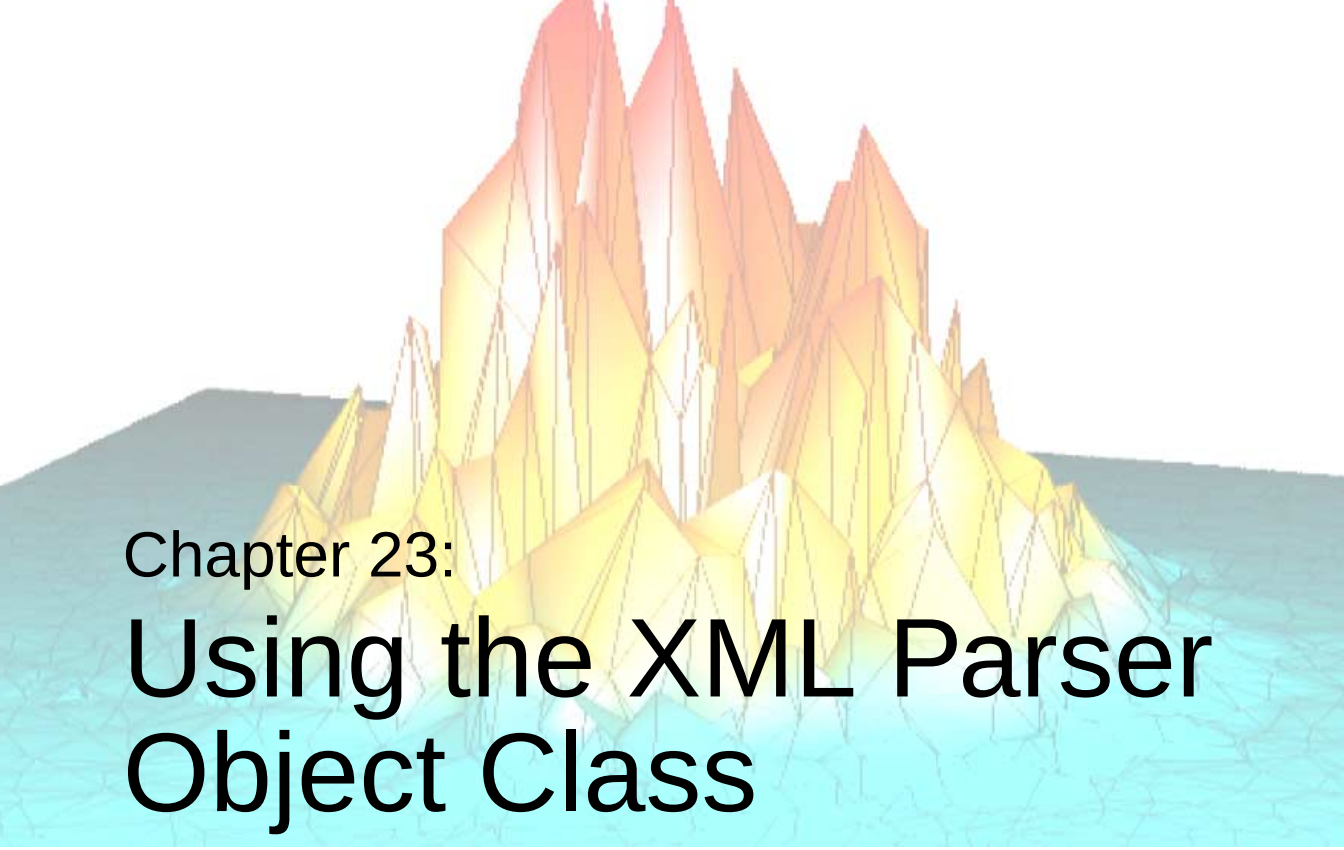
```
A -> class2::Print
```

IDL prints:

```
% CLASS2 is not a superclass of object class CLASS1.
% Execution halted at:  $MAIN$
```

Object Examples

We have included a number of examples of object-oriented programming as part of the IDL distribution. Many of the examples used in this volume are included — sometimes in expanded form — in the `examples/visual` 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" in the *IDL Reference Guide* manual for information on IDL's path.



Chapter 23: Using the XML Parser Object Class

The following topics are covered in this chapter:

About XML

XML (eXtensible Markup Language) provides a set of rules for defining semantic tags that can describe virtually any type of data in a simple ASCII text file. Data stored in XML-format files is both human- and machine-readable, and is often relatively easy to interpret either visually or programmatically. The structure of data stored in an XML file is described by either a Document Type Definition (DTD) or an XML schema, which can either be included in the file itself or referenced from an external network location.

It is beyond the scope of this manual to describe XML in detail. Numerous third-party books and electronic resources are available. The following texts may be useful:

- `http://www.w3.org` — information about many web standards, including XML related technologies.
- `http://www.w3schools.com` — tutorials on all manner of XML-related topics.
- `http://www.saxproject.org` — information about the Simple API for XML, the event-based XML parsing technology used by IDL.
- Brownell, David. *SAX2*. O'Reilly & Associates, 2002. ISBN: 0-596-00237-8.
- Harold, Eliotte Rusty. *XML Bible*. IDG Books Worldwide, 1999. ISBN: 0-7645-3236-7

About XML Parsers

There are two basic types of parsers for XML data:

- tree-based parsers
- event-based parsers.

Tree-based Parsers

Tree-based parsers map an XML document into a tree structure in memory, allowing you to select elements by navigating through the tree. This type of parser is generally based on the Document Object Model (DOM) and the tree is often referred to as a DOM tree.

Tree-based parsers are especially useful when the XML data file being parsed is relatively small. Having access to the entire data set at one time can be convenient and makes processing data based on multiple data values stored in the tree easy. However, if the tree structure is larger than will fit in physical memory or if the data

must be converted into a new (local) data structure before use, then tree-based parsers can be slow and cumbersome.

Event-based Parsers

Event-based parsers read the XML document sequentially and report parsing events (such as the start or end of an element) as they occur, without building an internal representation of the data structure. The most common examples of event-based XML parsers use the Simple API for XML (SAX), and are often referred to as a SAX parsers.

Event-based parsers allow the programmer to write *callback routines* that perform an appropriate action in response to an event reported by the parser. Using an event-based parser, you can parse very large data files and create application-specific data structures. The IDLffXMLSAX object class implements an event-based parser based on the SAX version 2 API.

Using the XML Parser

IDL's XML parser object class (IDLffXMLSAX) implements a SAX 2 event-based parser. The object's methods are a set of *callback routines* that are called automatically when the parser encounters different constituents of an XML document. For example, when the parser encounters the beginning of an XML element, it calls the `StartElement` method. When the `StartElement` method returns, the parser continues.

The IDLffXMLSAX object's methods are completely generic. As provided, they do nothing with the items encountered in the XML file. To use the parser object to read data from an XML file, you *must* write a subclass of the IDLffXMLSAX class, overriding the superclass's methods to accomplish your objectives. This requirement that you subclass the object makes the IDLffXMLSAX class unlike any other object class supplied by IDL.

For a detailed discussion of IDL object classes, subclassing, and method overriding, see Chapter 22, "Object Basics". For a description of the parser object class and its methods, see "IDLffXMLSAX" in the *IDL Reference Guide* manual.

Subclassing the IDLffXMLSAX Object Class

Writing a subclass of the IDLffXMLSAX object class is similar to writing a subclass of any of IDL's other object classes. The basic steps are:

1. Define a class structure for your subclass, inheriting from the IDLffXMLSAX object class.
2. Write methods to override the IDLffXMLSAX object class methods as necessary.
3. Write additional methods required for your application.
4. Create a class definition routine for your XML parser object.

Let's look at these steps individually:

Define a Class Structure

Every object class has a unique class structure that defines the instance data contained in the object. (See "Class Structures" on page 547 for details.) When writing your own parser object (a subclass of the IDLffXMLSAX object), you must first determine what instance data you need your parser object to contain, and define a class structure accordingly.

Note

Your parser object's class structure must inherit from the IDLffXMLSAX class structure. See "Inheritance" on page 549 for details.

For example, suppose you want to use your parser to extract an array of data from an XML file. You might choose to define your class structure to include an IDL pointer that will contain the data array. For this case, your class structure definition might look something like

```
void = {myParser, INHERITS IDLffXMLSAX, ptr:PTR_NEW()}
```

Within your subclass's methods, this data structure will always be available via the implicit `self` argument (see "Method Routines" on page 562 for details). Setting the value of `self.ptr` within a method routine sets the instance data of the object.

In most cases, your class structure definition will be included in a routine that does *Automatic Structure Definition* (see "Automatic Class Structure Definition" on page 548 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" on page 566 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 579 and "Example: Reading Data Into Structures" on page 586 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 574, 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 579 and "Example: Reading Data Into Structures" on page 586 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" in the *IDL Reference Guide* manual for details.

Example: Reading Data Into an Array

This example subclasses the IDLffXMLSAX parser object class to create an object class named `xml_to_array`. The `xml_to_array` object class is designed to read numerical values from an XML file with the following structure:

```
<array>
  <number>0</number>
  <number>1</number>
  ...
</array>
```

and place those values into an IDL array variable.

Note

This example is a very simple example. It is designed to illustrate how an event-based XML parser is constructed using the IDLffXMLSAX object class. An application that reads real data from an XML file will most likely be quite a bit more complicated.

Creating the xml_to_array Object Class

In order to read the XML file and return an array variable, we will need to create an object class definition that inherits from the IDLffXMLSAX object class, and override the following superclass methods: `Init`, `Cleanup`, `StartDocument`, `Characters`, `StartElement`, and `EndElement`. Since this example does not retrieve data using any of the other IDLffXMLSAX methods, we do not need to override those methods. In addition, we will create a new method that allows us to retrieve the array data from the object instance data.

Note

This example is included in the file `xml_to_array__define.pro` in the `examples/data_access` subdirectory of the IDL distribution.

Object Class Definition

The following routine is the definition of the `xml_to_array` object class:

```
PRO xml_to_array__define

void = {xml_to_array, $
   INHERITS IDLffXMLSAX, $
   charBuffer:'', $
```

```
    pArray:PTR_NEW()}
END
```

The following items should be considered when defining this class structure:

- The structure definition uses the INHERITS keyword to inherit the object class structure and methods of the IDLffXMLSAX object.
- The `charBuffer` structure field is set equal to an empty string.
- The `pArray` structure field is set equal to an IDL pointer. We will use this pointer to store the numerical array data we retrieve.
- The routine name is created by adding the string "`__define`" (note the *two* underscore characters) to the class name.

Why do we store the array data in a pointer variable? Because the fields of a named structure (`xml_to_array`, in this case) must always contain the same type of data as when that structure was defined. Since we want to be able to add values to the data array as we parse the XML file, we will need to extend the array with each new value. If we began by defining the size of the array in the structure variable, we would not be able to extend the array. By holding the data array in a pointer, we can extend the array without changing the format of the `xml_to_array` object class structure.

Note

Although we describe this routine first here, the `xml_to_array__define` routine must be the *last* routine in the `xml_to_array__define.pro` file.

Init Method

The `Init` method is called when the an `xml_to_array` parser object is created by a call to OBJ_NEW. The following routine is the definition of the `Init` method:

```
FUNCTION xml_to_array::Init
  self.pArray = PTR_NEW(/ALLOCATE_HEAP)
  RETURN, self -> IDLffxmlsax::Init()
END
```

We do two things in this method:

- We initialize the pointer in the `pArray` field of the class structure variable.

  Note

  Within a method, we can refer to the class structure variable with the implicit parameter `self`. Remember that `self` is actually a reference to the `xml_to_array` object instance.

- The return value from this function is the return value of the superclass's `Init` method, called on the `self` object reference.

Note

The initialization task (setting the value of the `pArray` field) is performed before calling the superclass's `Init` method.

See "IDLffXMLSAX::Init" in the *IDL Reference Guide* manual for details on the method we are overriding.

Cleanup Method

The `Cleanup` method is called when the `xml_to_array` parser object is destroyed by a call to OBJ_DESTROY. The following routine is the definition of the `Cleanup` method:

```
PRO xml_to_array::Cleanup

IF (PTR_VALID(self.pArray)) THEN PTR_FREE, self.pArray

END
```

All we do in the `Cleanup` method is to release the `pArray` pointer, if it exists.

See "IDLffXMLSAX::Cleanup" in the *IDL Reference Guide* manual 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" in the *IDL Reference Guide* manual 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" in the *IDL Reference Guide* manual 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" in the *IDL Reference Guide* manual 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" in the *IDL Reference Guide* manual for details on the method we are overriding.

Note

In both the `StartElement` and `EndElement` methods, we rely on the validity of the XML data file. Our CASE statements only need to handle the element types described in the XML file's DTD or schema (in this case, the only elements are `<array>` and `<number>`). We do not need an ELSE clause in the CASE statement. If an unknown element is found in the XML file, the parser will report a validation error.

GetArray Method

The `GetArray` method allows us to retrieve the array data stored in the `pArray` pointer variable. The following routine is the definition of the `GetArray` method:

```
FUNCTION xml_to_array::GetArray

IF (N_ELEMENTS(*self.pArray) GT 0) THEN $
   RETURN, *self.pArray $
ELSE RETURN , -1

END
```

Here, we check to see whether the array pointed at by `pArray` contains any data. If it does contain data, we return the array. If the array contains no data, we return the value `-1`.

Using the xml_to_array Parser

To see the `xml_to_array` parser in action, you can parse the file `num_array.xml`, found in the `examples/data` subdirectory of the IDL distribution. This `num_array.xml` file contains the fragment of XML like the one shown in the beginning of this section, and includes 20 extra `<number>` elements. The `num_array.xml` file also includes a DTD describing the structure of the file.

Enter the following statements at the IDL command line:

```
xmlObj = OBJ_NEW('xml_to_array')
xmlFile = FILEPATH('num_array.xml', $
   SUBDIRECTORY = ['examples', 'data'])
xmlObj -> ParseFile, xmlFile
myArray = xmlObj -> GetArray()
OBJ_DESTROY, xmlObj
HELP, myArray
PRINT, myArray
```

IDL prints:

```
MYARRAY          INT       = Array[20]
 0   1   2   3   4   5   6   7   8   9   10   11
12  13  14  15  16  17  18  19
```

Example: Reading Data Into Structures

This example subclasses the IDLffXMLSAX parser object class to create an object class named `xml_to_struct`. The `xml_to_struct` object class is designed to read data from an XML file with the following structure:

```
<Solar_System>
   <Planet NAME='Mercury'>
      <Orbit UNITS='kilometers' TYPE='ulong64'>579100000</Orbit>
      <Period UNITS='days' TYPE='float'>87.97</Period>
      <Satellites TYPE='int'>0</Satellites>
   </Planet>
  ...
</Solar_System>
```

and place those values into an IDL array containing one structure variable for each `<Planet>` element. We use a structure variable for each `<Planet>` element so we can capture data of several data types in a single place.

Note

While this example is more complicated than the previous example, it is still rather simple. It is designed to illustrate a method whereby more complex XML data structures can be represented in IDL.

Creating the xml_to_struct Object Class

To read the XML file and return a structure variable, we will need to create an object class definition that inherits from the IDLffXMLSAX object class, and override the following superclass methods: `Init`, `Characters`, `StartElement`, and `EndElement`. Since this example does not retrieve data using any of the other IDLffXMLSAX methods, we do not need to override those methods. In addition, we will create a new method that allows us to retrieve the structure data from the object instance data.

Notice that the elements of the XML data file include *attributes*. While we will retrieve and use some of the attribute data from the file, we will ignore some of it.

Note

When parsing an XML data file, you can pick and choose the data you wish to pull into IDL. This ability to selectively retrieve data from the XML file is one of the great advantages of an event-based parser over a tree-based parser.

Note

This example is included in the file `xml_to_struct__define.pro` in the `examples/data_access` subdirectory of the IDL distribution.

Object Class Definition

The following routine is the definition of the `xml_to_struct` object class:

```
PRO xml_to_struct__define

void = {PLANET, NAME: "", Orbit: 0ull, period:0.0, Moons:0}
void = {xml_to_struct, $
   INHERITS IDLffXMLSAX, $
   CharBuffer:"", $
   planetNum:0, $
   currentPlanet:{PLANET}, $
   Planets : MAKE_ARRAY(9, VALUE = {PLANET})}

END
```

The following items should be considered when defining this class structure:

- Before creating the object class structure, we define a structure named PLANET. We will use the PLANET structure to store data from the `<Planet>` elements of the XML file.
- The object class structure definition uses the `INHERITS` keyword to inherit the object class structure and methods of the IDLffXMLSAX object.
- The `charBuffer` structure field is set equal to a string value. We will use this field to accumulate character data stored in XML elements.
- The `planetNum` structure field is set equal to an integer value. We will use this field to keep track of which array element we are currently populating.
- The `currentPlanet` structure field is set equal to a PLANET structure.
- The `Planets` structure field is set equal to a nine-element array of PLANET structures.
- The routine name is created by adding the string "`__define`" (note the *two* underscore characters) to the class name.

We have explicitly defined our `Planets` structure field as a nine-element array of PLANET structures, which we can do because we know exactly how many `<Planet>` elements will be read from our XML file. Specifying the exact size of the data array in the class structure definition is very efficient (since we create the array

only once) and eliminates the need to free the pointer in the `Cleanup` method. However, it has the following consequences:

- We must explicitly keep track of the index of the array element we are populating, and increment it after we have finished with a given element (see the `EndElement` method below).
- We must know in advance how many elements the array will hold. If the size of the final array is unknown, it is more efficient to use a pointer to an array, as we did in the previous example, and allow the array to grow as elements are added. See "Building Complex Data Structures" on page 593 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" in the *IDL Reference Guide* manual 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" in the *IDL Reference Guide* manual 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" in the *IDL Reference Guide* manual 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" in the *IDL Reference Guide* manual 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 579, 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.

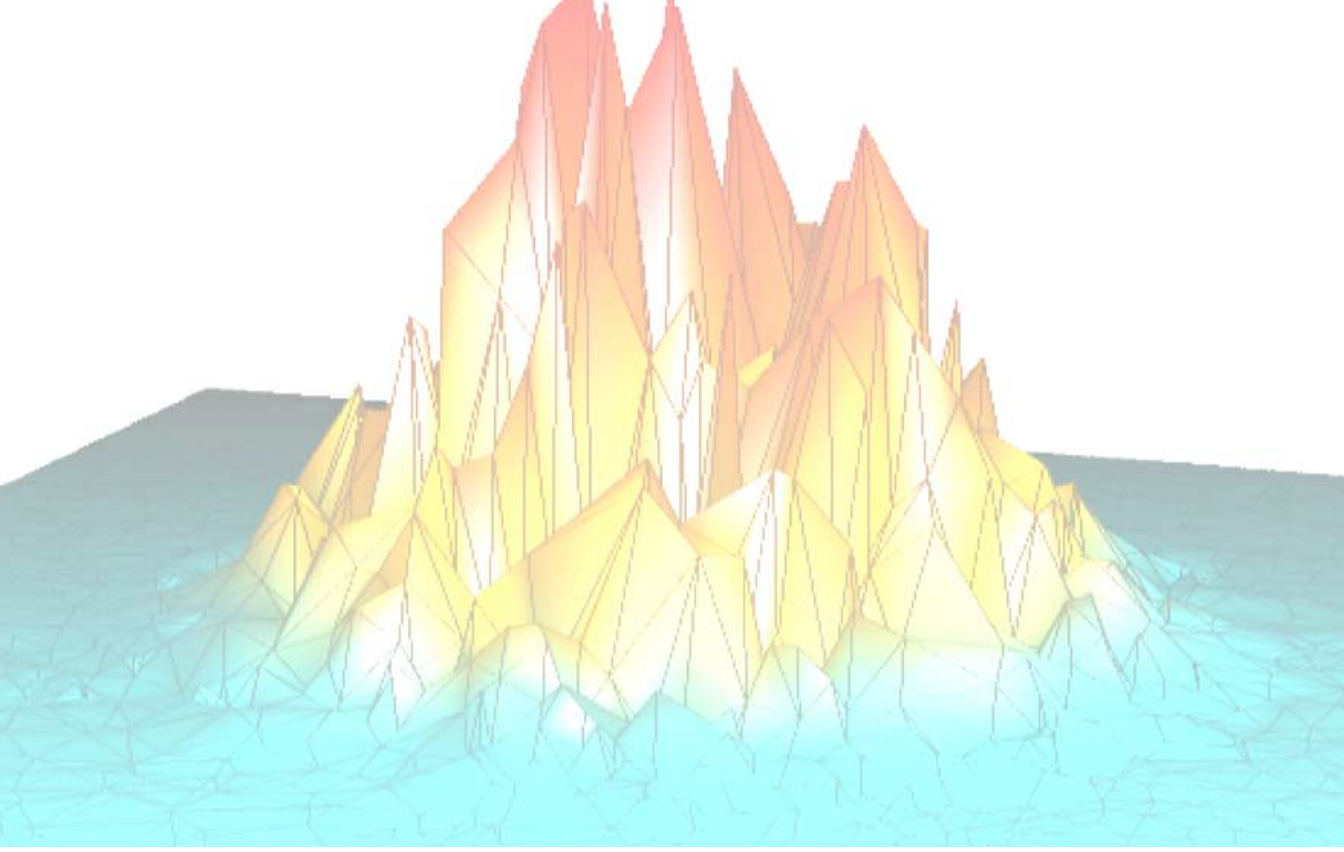
In "Example: Reading Data Into Structures" on page 586, we illustrated the technique of using a pre-defined array to store our instance data.

Using Nested Structures

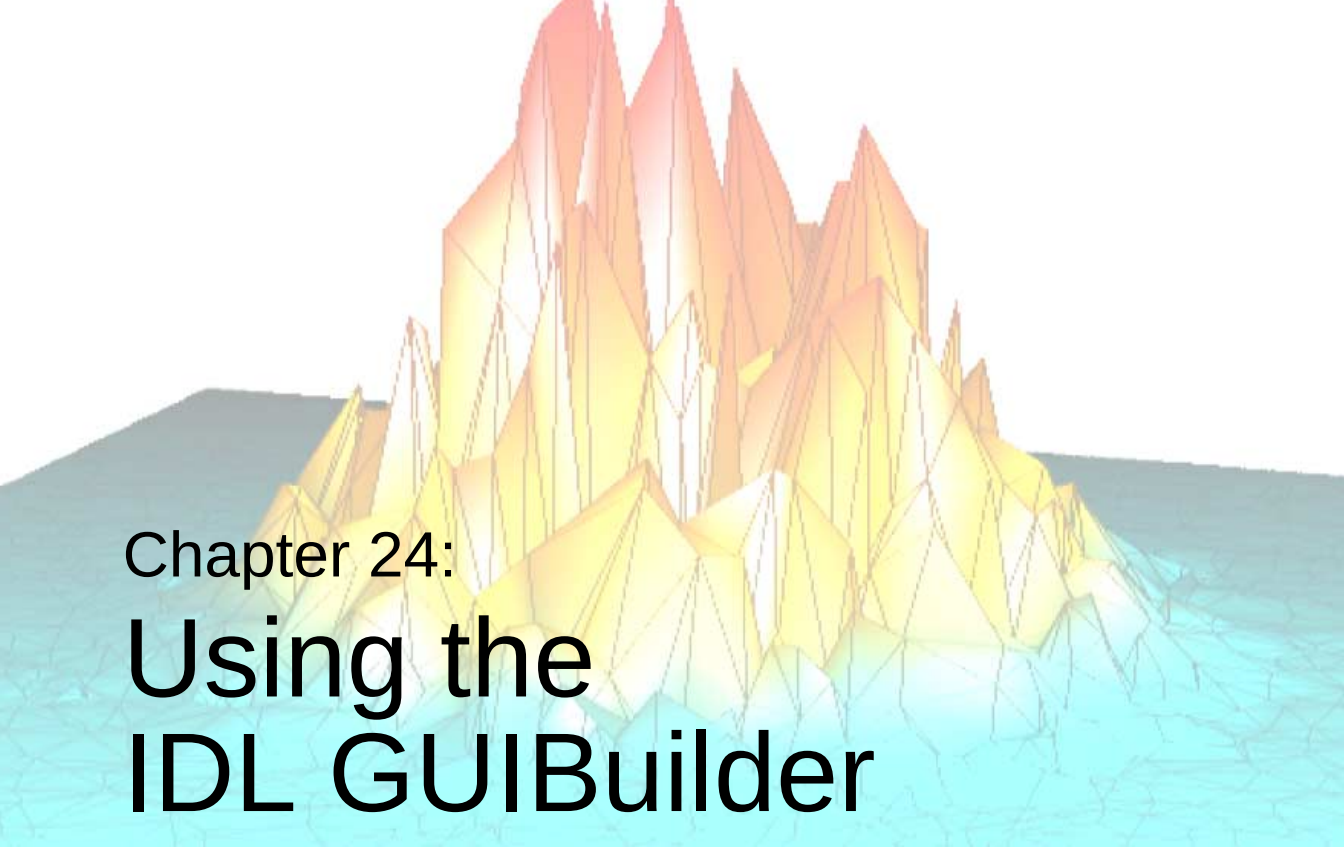
If your data structure is complex, you may be inclined to represent your data as a set of nested IDL structure variables. While nesting structure variables can help you create a data structure that emulates the structure of your XML file, deeply nested structures can make your code more difficult to 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.



Part V: Creating Graphical User Interfaces in IDL



Chapter 24:
Using the IDL GUIBuilder

The following topics are covered in this chapter:

Overview

The IDL GUIBuilder is part of the IDLDE for Windows. The IDL GUIBuilder supplies you with a way to interactively create user interfaces and then generate the IDL source code that defines that interface and contains the event-handling routine place holders.

Note

The IDL GUIBuilder is supported on Windows only. However, the code it generates is portable and runs on all IDL supported platforms. Since applications built with IDL GUIBuilder may require functionality added in the current release, generated code only runs on the version of IDL you generated the code on or greater.

The IDL GUIBuilder has several tools that simplify application development. These tools allow you to create the widgets that make up user interfaces, define the behavior of those widgets, define menus, and create and edit color bitmaps for use in buttons.

Note

When using code generated by the IDL GUIBuilder on other non-Windows platforms, more consistent results are obtained by using a row or column layout for your bases instead of a bulletin board layout. By using a row or column layout, problems caused by differences in the default spacing and decorations (e.g., beveling) of widgets on each platform can be avoided

These are the basic steps you will follow when building an application interface using the IDL GUIBuilder:

1. Interactively design and create a user interface using the components, or *widgets*, supplied in the IDL GUIBuilder. Widgets are simple graphical objects supported by IDL, such as sliders or buttons.
2. Set attributes for each widget. The attributes control the display, initial state, and behavior of the widget.
3. Set event properties for each widget. Each widget has a set of events to which it can respond. When you design and create an application, it is up to you to decide if and how a widget will respond to the events it can generate. The first step to having a widget respond to an event is to supply an event procedure name for that event.
4. Save the interface design to an IDL resource file, `*.prc` file, and generate the portable IDL source code files. There are two types of generated IDL source

code: widget definition code (`*.pro` files) and event-handling code (`*_eventcb.pro` files).

5. Modify the generated `*_eventcb.pro` event-handling code file using the IDLDE, then compile and run the code. This code can run on any IDL-supported platform.

The `*_eventcb.pro` file contains place holders for all of the event procedures you defined for the widgets, and you complete the file by filling in the necessary event callback routines for each procedure.

Warning

Once you have generated the widget definition code (`*.pro` files), you should not modify this file manually. If you decide to change your interface definition, you will need to regenerate the interface code, and will therefore overwrite that `*.pro` file. Any new event handling code will not be overwritten but will instead be appended.

For information about IDL widgets, and how to create user interfaces programmatically (without the IDL GUIBuilder), see Chapter 25, “Widgets”.

Starting the IDL GUIBuilder

To open a new IDL GUIBuilder window, do one of the following:

- Select **File** → **New** → **GUI** from the IDLDE menu.
- Click the “New GUI” button on the IDLDE toolbar.

Each of these actions opens a new IDL GUIBuilder window and displays the IDL GUIBuilder toolbar. The IDL GUIBuilder window contains a top-level base widget, as shown in the following figure. This top-level base holds all of the widgets for an individual interface; it is the top-level parent in the widget hierarchy being created.

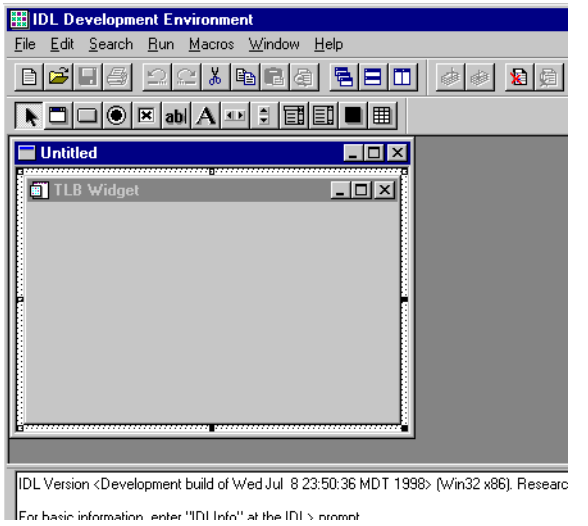


Figure 24-1: IDLDE with IDL GUIBuilder Window

Opening Existing Interface Definitions

To open an existing interface design in the IDL GUIBuilder:

1. Do one of the following to launch the Open dialog:
   - Select **File** → **Open** from the IDLDE menu.
   - Click on the “Open” button on the IDLDE toolbar.
2. In the Open dialog, select the appropriate `*.prc` file, and click Open.

The `*.prc` portable resource file contains the widget definitions that make up the widget hierarchy and define your interface design. When you click Open, the existing definition is displayed in an IDL GUIBuilder window. You can modify the interface then save it, and you can generate new IDL source code for the modified definition.

Creating an Example Application

The following example takes you through the process of creating your first application with the IDL GUIBuilder and the IDLDE. You will create the user interface and write the event callback routines.

This simple example application contains a menu and a draw widget. When complete, the running application allows the user to open and display a graphics file in PNG format, change the color table for the image display, and perform smooth operations on the displayed image.

This example introduces you to some of the basic procedures you will use to create applications with the IDL GUIBuilder; it shows you how to define menus, create widgets, set widget properties, and write IDL code to handle events.

Defining Menus for the Top-level Base

To define the menu, follow these steps:

1. Open a new IDL GUIBuilder window by selecting **File → New → GUI** from the IDLDE menu, or click the “New GUI” button on the IDLDE toolbar.
2. Drag out the window then the top-level base to a reasonable size for displaying an image. For example, drag the base out so that it has an X Size attribute value of 500 and a Y Size attribute value of 400. To view the attribute values, right-click on the base, and choose Properties from menu. In the Properties dialog, scroll down to view the X Size and Y Size attribute values.
3. Right-click on the top-level base in the IDL GUIBuilder window, then choose **Edit Menu**. This opens the Menu Editor.
4. In the Menu Editor Menu Caption field, enter “File” and click Insert to set the entered value and add a new line after the currently selected line. The new line becomes the selected line.
5. To define the File menu items, do the following:
   A. With the new line selected, click on the right arrow in the Menu Editor, which indents the line and makes it a menu item.
   B. Click in the Menu Caption field and enter “Open...”.
   C. Click in the Event Procedure field and enter “OpenFile”. The OpenFile routine will be called when the user selects this menu.

D. To create a separator after the Open menu, click the line button at the right side of the dialog (above the arrow buttons).

E. To set the values and move to a new line, click Insert.

F. In the Menu Caption field, enter “Exit”.

G. In the Event Procedure field, enter “OnExit”.

H. To set the values and move to a new line, click Insert.

6. To define the Tools menu and its one item, do the following:

   A. With the new line selected, click the left arrow to make the line a top-level menu.

   B. In the Menu Caption field, enter “Tools”, then click Insert.

   C. Click the right arrow to make the new line a menu item.

   D. In the Menu Caption field, enter “Load Color Table”.

   E. In the Event Procedure field, enter “OnColor”.

   F. To set the values and move to a new line, click Insert.

7. To define the Analyze menu and its one menu item, do the following:

   A. With the new line selected, click the left arrow to make the line a top-level menu.

   B. In the Menu Caption field, type “Analyze”, then press Enter.

   C. Click the right arrow to make the new line a menu item.

   D. In the Menu Caption field, enter “Smooth”.

   E. In the Event Procedure field, enter “DoSmooth”.

Your entries should look like those shown in the following figure.

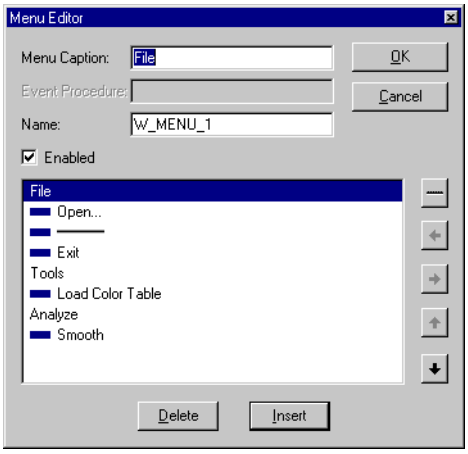


Figure 24-2: Menu Editor Dialog with Example Menus

8. Save your menu definitions by clicking OK in the Menu Editor.

Note

For more information about using the Menu Editor, see "Using the Menu Editor" on page 621.

9. At this time you can click on the menus to test them. Your interface should look similar to the one in the figure below.
10. Select **File** → **Save** from the IDLDE menu, which opens the "Save As" dialog.
11. In the "Save As" dialog, select a location, enter "example.prc" in the File name field, and click Save. This writes the portable resource code to the specified file.

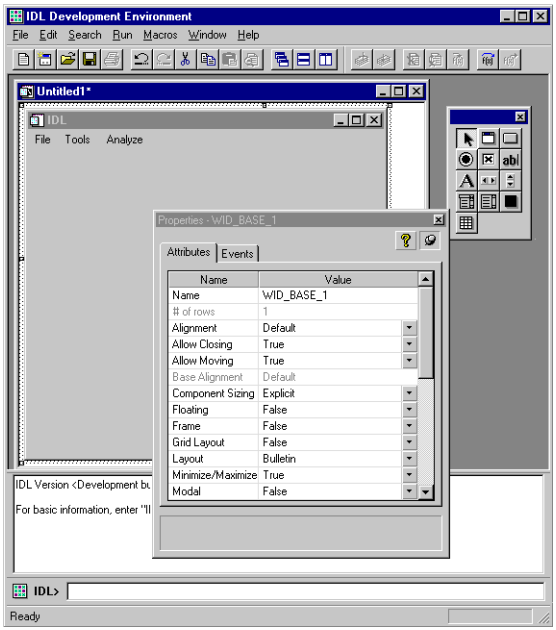


Figure 24-3: IDL GUIBuilder with Example Application

Creating a Draw Widget

To create a draw area that will display PNG image files, follow these steps:

1. Click on the Draw Widget tool button, then drag out an area that fills the top-level base display area. Leave a small margin around the edge of the draw area when you drag it out.
2. Right click on the draw area, and choose **Properties** to open the Properties dialog for the draw area.
3. In the Properties dialog, click the push pin button so the dialog will stay open and on top.
4. In the Properties dialog, change the draw widget Name attribute value to “Draw”.

Later, you will write code to handle the display of the image in this draw area widget. Renaming the widget now will make it easier to write the code later; the “Draw” name is easy to remember and to type.

Note

The Name attribute must be unique to the widget hierarchy.

5. In the IDL GUIBuilder window, click on the top-level base widget to select it. When you do so, the Properties dialog will update and display the attributes for this base widget.
6. In the Properties dialog, change the base widget Component Sizing attribute to Default. This sizes the base to the draw widget size you created.

   When you first dragged out the size of the base, the Component Sizing attribute changed from Default to Explicit—you explicitly sized the widget. Now that the base widget contains items, you can return it to Default sizing, and IDL will handle the sizing of this top-level base.
7. In the Properties dialog, change the base widget Layout attribute to Column.
8. Select **File** → **Save** to save your new modifications to the `example.prc` file. The application should look like the one shown in the following figure.

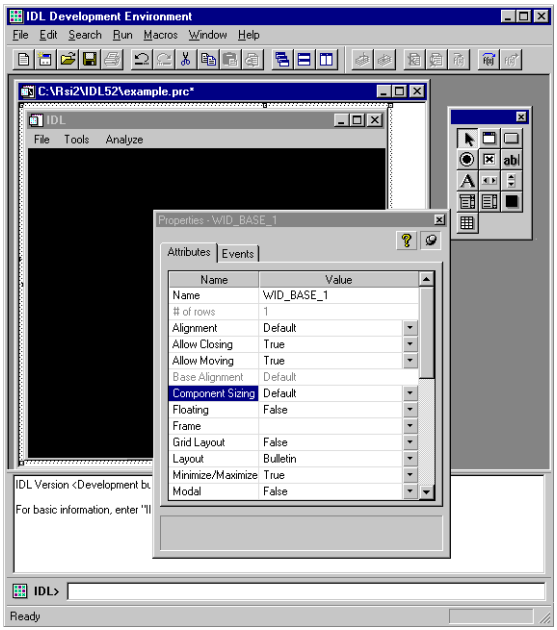


Figure 24-4: Complete Example Application

Running the Application in Test Mode

You can run the application in test mode, which allows you to test the display of widgets and menus. To run your application in test mode, do one of the following:

- Select **Run** → **Test GUI** from the IDLDE menu.
- Press Control+t.

Both these actions display the interface as it will look when it runs. You can click on the menus, but there is no active event handling in test mode.

To exit test mode, do one of the following:

- Press the Esc key.
- Click the X in the upper-right corner of the test application window.

Generating the IDL Code

To generate the code for the example application, follow these steps:

1. Select **File → Generate .pro**. In the "Save As" dialog, find the location where you want the files saved, enter "example.pro" in the File name field, and click Save. This generates an `example.pro` widget definition file and an `example_eventcb.pro` event-handling file.

   The `example.pro` file contains the widget definition code, and you should never modify this file. If you decide later to change your interface, you will need to regenerate this interface code, and thus overwrite the widget code file.

   The `example_eventcb.pro` contains place holders for all the event procedures you defined in the IDL GUIBuilder Menu Editor and Properties dialog. You must complete these event procedures by filling in event callback routines. If you generate code after you have modified this file, any new event handling code will not be overwritten but will instead be appended. For information on ways to handle regenerating the `*_eventcb.pro` file, see "Notes on Generating Code a Second Time" on page 632.

   For more information on interface definitions and generated code, see "Generating Files" on page 631.

Note

You should modify *only* the generated event-handling file (`*_eventcb.pro`); you should never modify the generated interface code (the `*.pro` file).

Handling the Open File Event

You can now modify the generated `example_eventcb.pro` file to handle the events for the application. First, you will modify the OpenFile routine.

When the user selects Open from the File menu of the example application, the appropriate event structure is sent, and the OpenFile routine handles the event. For this application, the Open menu item will launch an Open dialog to allow the user to choose a PNG file, and then the routine will check the selected file's type, read the image, and display it in the draw area.

To open the file and add the code to handle the OpenFile event, follow these steps:

1. Select **File → Open** from the IDLDE menu. In the "Open" dialog, select the `example_eventcb.pro` file, and click Open. This file contains the event handling routine place holders, which you will now complete.

2. In the `example_eventcb.pro` file, locate the OpenFile procedure, which looks like this:

```
pro OpenFile, Event

end
```

Tip

To easily find the OpenFile routine, select OpenFile from the Functions/Procedures drop-down list on the IDLDE toolbar.

3. Add the following code between the PRO and END statements to handle the event:

```
; If there is a file, draw it to the draw widget.
sFile = DIALOG_PICKFILE(FILTER='*.png')
IF(sFile NE "")THEN BEGIN
   ; Find the draw widget, which is named Draw.
   wDraw = WIDGET_INFO(Event.top, FIND_BY_UNAME='Draw');
   ; Make sure something was found.
   IF(wDraw GT 0)THEN BEGIN
      ; Make the draw widget the current, active window.
      WIDGET_CONTROL, wDraw, GET_VALUE=idDraw
      WSET,idDraw
      ; Read in the image.
      im = READ_PNG(sFile, r, g, b)
      ; If TrueColor image, quantize image to pseudo-color:
      IF (SIZE(im, /N_DIM) EQ 3) THEN $
         im = COLOR_QUAN(im, 1, r, g, b)
      ; Size the image to fill the draw area.
      im = CONGRID(im, !D.X_SIZE, !D.Y_SIZE)
      ; Handle TrueColor displays:
      DEVICE, DECOMPOSED=0
      ; Load color table, if one exists:
      IF (N_ELEMENTS(r) GT 0) THEN TVLCT, r, g, b
      ; Display the image.
      TV, im
      ; Save the image in the uvalue of the top-level base.
      WIDGET_CONTROL, Event.top, SET_UVALUE=im, /NO_COPY
   ENDIF
ENDIF
```

Note

In the added code, you used the FIND_BY_UNAME keyword to find the draw widget using its name attribute. In this example, the widget name, “Draw”, is the one you gave the widget in the IDL GUIBuilder Properties dialog. The widget name is case-sensitive.

Handling the Exit Event

To add the code that causes the example application to close when the user chooses Exit from the File menu, follow these steps:

1. Locate the OnExit routine place holder, which looks like this:

```
pro OnExit, Event

end
```

2. add the following statement between the PRO and END statements to handle the destruction of the application:

```
WIDGET_CONTROL, Event.top, /DESTROY
```

Handling the Load Color Table Event

To add the code that causes the example application to open the IDL color table dialog when the user chooses Load Color Table from the Tools menu, follow these steps:

1. Locate the OnColor routine place holder, which looks like this:

```
pro OnColor, Event

end
```

2. Add the following code between the PRO and END statements:

```
XLOADCT, /BLOCK
; Find the draw widget, which is named Draw:
wDraw = WIDGET_INFO(Event.top, FIND_BY_UNAME='Draw')
IF(wDraw GT 0) THEN BEGIN
   ; Make the draw widget the current, active window:
   WIDGET_CONTROL, wDraw, GET_VALUE=idDraw
   WSET, idDraw
   WIDGET_CONTROL,Event.top, GET_UVALUE=im, /NO_COPY
   ; Make sure the image exists:
   IF (N_ELEMENTS(im) NE 0) THEN BEGIN
   ; Display the image:
      TV, im
      ; Save the image in the uvalue of the top-level base:
      WIDGET_CONTROL, Event.top, SET_UVALUE=im, /NO_COPY
   ENDIF
ENDIF
```

This procedure opens a dialog from which the user can select from a set of predefined color tables. When the user selects a color table, it is loaded and the displayed image changes accordingly.

Handling the Smooth Event

When the user selects **Smooth** from the **Analyze** menu, a smooth operation is performed on the displayed image. The smooth operation displays a smoothed image with a boxcar average of the specified width, which in the example code is 5.

To add the callback routines to handle the smooth operation, follow these steps:

1. Locate the DoSmooth routine place holder, which looks like this:

```
pro DoSmooth, Event

end
```

2. Add the following code between the PRO and END statements to handle the smooth operation:

```
; Get the image stored in the uvalue of the top-level-base.
WIDGET_CONTROL, Event.top, GET_UVALUE=image, /NO_COPY
; Make sure the image exists.
IF(N_ELEMENTS(image) GT 0)THEN BEGIN
   ; Smooth the image.
   image = SMOOTH(image, 5)
   ; Display the smoothed image.
   TV, image
   ; Place the new image in the uvalue of the button widget.
   WIDGET_CONTROL, Event.top, SET_UVALUE=image, /NO_COPY
ENDIF
```

3. Select **File** → **Save**, to save all your changes to the `example_eventcb.pro` file.

Compiling and Running the Example Application

To compile and run your example application, type `example` at the IDL> command prompt. The following figure shows the example application and the IDL color table dialog.

In the running application, you can open and display a PNG file. Then, you can open the XLOADCT dialog and change the color table used in displaying the image, or you can perform the smooth procedure on the image.

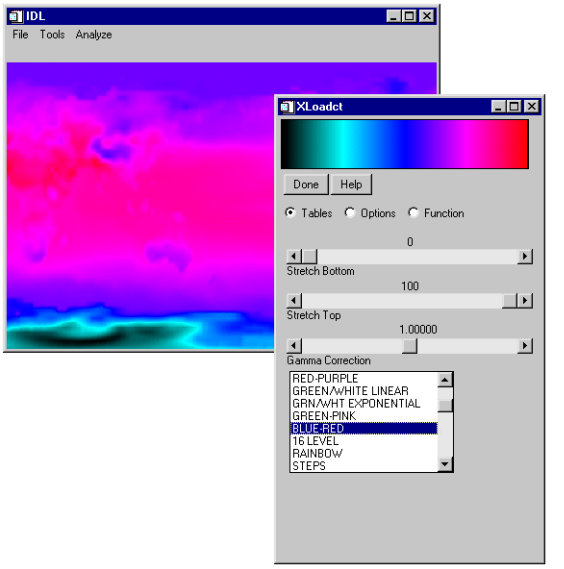


Figure 24-5: Running Example Application and XLOADCT Dialog

IDL GUIBuilder Tools

You will use the following tools to design and construct a graphical interface using the IDL GUIBuilder:

- The IDL GUIBuilder Toolbar, which you use to create the widgets that make up your interface. See "Using the IDL GUIBuilder Toolbar" on page 614 and "Widget Operations" on page 628.
- The Widget Properties dialog, which you use to set widget attributes and event properties. See "Using the Properties Dialog" on page 617 and "Widget Properties" on page 647.
- The Widget Browser, which you can use to see the widget hierarchy and to modify certain aspects of the widgets in your application. See "Using the Widget Browser" on page 619.
- The Menu Editor, which you use to define menus to top-level bases and buttons. See "Using the Menu Editor" on page 621.
- The Bitmap Editor, which you use to create or modify bitmap images to be displayed on button widgets. See "Using the Bitmap Editor" on page 624.
- The IDLDE to modify, compile, and run the generated code (see Chapter 2, "The IDL Development Environment" in the *Using IDL* manual.

Using the IDL GUIBuilder Toolbar

The IDL GUIBuilder has its own toolbar in the IDE, which you use to create the widgets for your user interface. The following figure shows the toolbar.

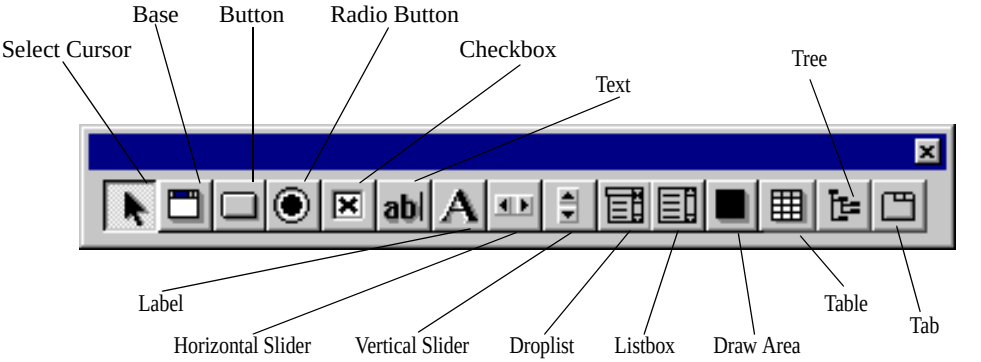


Figure 24-6: IDL GUIBuilder Toolbar

These are the widget types you can create using the IDL GUIBuilder toolbar:

Widget	Description
Base	Creates a container for a group of widgets within a top-level base container. A top-level base is contained in the IDL GUIBuilder window, and you build your interface in it. Use base widgets within the top-level base to set up the widget hierarchy, layout, and to organize the application. For example, you can use a base widget to group a set of buttons. For information on base properties, see "Base Widget Properties" on page 654.
Button	Creates a push button. The easiest way to allow a user to interact with your application is through a button click. You can have button widgets display labels, menus, or bitmaps. For information on button properties, see "Button Widget Properties" on page 666.

Table 24-1: Widget Types

Widget	Description
Radio Button	Creates a toggle button that is always grouped within a base container. Use radio buttons to present a set of choices from which the user can pick only one. For information on radio button properties, see "Button Widget Properties" on page 666.
Checkbox	Creates a checkbox, which you can use either as a single toggle button to indicate a particular state is on or off or as a list of choices from which the user can select none to all choices. Checkboxes are created within a base container. For information on checkbox properties, see "Button Widget Properties" on page 666.
Text	Creates a text widget. Use text widgets to get input from users or to display multiple lines of text. For information on text widget properties, see "Text Widget Properties" on page 671.
Label	Creates a label. Use label widgets to identify areas of your application or to label widgets that do not have their own label property. Use labels when you have only a single line of text and you do not want the user to be able to change the text. For information on label widget properties, see "Label Widget Properties" on page 676.
Horizontal and Vertical Sliders	Creates a slider with a horizontal or vertical layout. Use slider widgets to allow the user to control program input, such as adjust the speed of movement for a rotating image. For information on slider properties, see "Slider Widget Properties" on page 678.
Droplist	Creates a droplist widget, which you can use to present a scrollable list of items for the user to select from. The droplist is an effective way to present a lot of choices without using too much interface space. For information on droplist properties, see "Droplist Widget Properties" on page 680.
Listbox	Creates a list widget, which you can use to present a scrollable list of items for the user to select from. For information on listbox properties, see "Listbox Widget Properties" on page 682.

Table 24-1: (Continued) Widget Types

Widget	Description
Draw Area	Creates a draw area, which you can use to display graphics in your application. The draw area can display IDL Direct Graphics or IDL Object Graphics, depending on how you set its properties. For information on the draw area properties, see "Draw Widget Properties" on page 685.
Table	Creates a table widget, which you can use to display data in a row and column format. You can allow users to edit the contents of the table. For information on the table widget properties, see "Table Widget Properties" on page 692.
Tab	Creates a tab widget on which different "pages" (base widgets and their children) can be displayed by selecting the appropriate tab. For information on the tab widget properties, see "Tab Widget Properties" on page 700.
Tree	Creates a tree widget, which presents a hierarchical view that can be used to organize a wide variety of data structures and information. For information on the tree widget properties, see "Tree Widget Properties" on page 702.

Table 24-1: (Continued) Widget Types

Note

The Select Cursor button returns the cursor to its standard state, and it indicates that the cursor is in that state. After you click on another button and create the selected widget, the cursor returns to the selection state.

Creating Widgets

All widgets for a user interface must be descendents of a top-level base; in the IDL GUIBuilder window, all widgets must be contained in a top-level base widget. When you open an IDL GUIBuilder window, it contains a top-level base. You can add base widgets to that top-level widget to form a widget hierarchy. The added bases can act as containers for groups of widgets.

To create a widget, do one of the following:

- Click on the appropriate button on the toolbar, then drag out an area within the top-level base widget. When you release the mouse button, a widget the size of the dragged-out area is created.

- Click on the appropriate button on the toolbar, then click within the top-level base area. This creates a widget of the default size.

After you add widgets to a top-level base, you can resize, move, and delete them, and you can change their parent base. You can also set properties for each widget. For information on how to operate on widgets, see "Widget Operations" on page 628, and for information on setting properties, see "Using the Properties Dialog" on page 617.

Using the Properties Dialog

For each widget, you can define attribute and event procedure properties. A widget's attributes define how it will display on the screen and its basic behaviors. The attributes you can set for a selected widget are displayed on the Attributes tab of the Properties dialog. These attributes are initially set to default values.

Event procedures are the predefined set of events a widget can recognize. When you write an application, you decide if and how the widget will respond to each of the possible events. The events that a selected widget recognizes are displayed on the Events tab of the Properties dialog. The event values are initially undefined. Supply event routine names for only those events to which you want the application to respond.

Opening the Properties dialog

To open the Properties dialog for a widget, do one of the following:

- Right-click on the widget in the IDL GUIBuilder window, and choose **Properties** from the menu.
- Select the widget, and choose **Properties** from the **Edit** menu.

These actions open a Properties dialog similar to the one shown in the following figure.

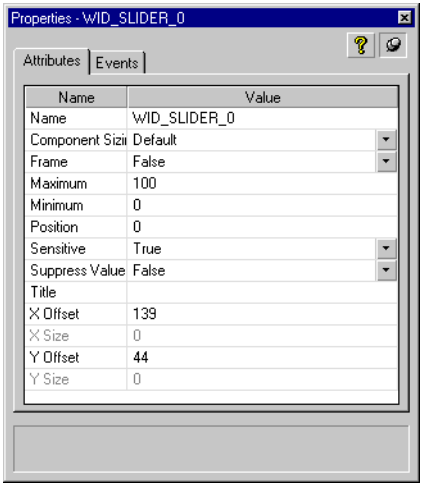


Figure 24-7: Properties Dialog for a Slider Widget

The status area at the bottom of the Properties dialog contains a description of the currently selected attribute or event. In addition, for each property that maps directly to an IDL keyword, there is a tool-tip that provides the name of the IDL keyword.

To display a tool-tip, place the cursor over the property name. The tool-tips are displayed only for properties that map to IDL keywords.

Note

If you have multiple widgets selected in the IDL GUIBuilder window, the Properties dialog displays the properties for the primary selection, which is indicated by the darker, filled-in sizing handles around the widget. When you select multiple widgets, only one is marked as the primary selection.

To keep the Properties dialog on top, click the push pin button.

The Properties dialog will close as soon as it loses focus, unless you click the push pin button. If you click the push pin button, the Properties dialog stays on top and updates to reflect the properties of the currently selected widget.

To close the Properties dialog when the push pin is being used, do one of the following:

- Click the push pin again, and the dialog will close when it loses focus.

- Press Escape while the dialog has focus.
- Click the X in the upper right corner of the dialog.

Any changes you make to values in the Properties dialog are automatic; you will see the results of all visual changes immediately. For example, any changes you make to the alignment or column setting will change the layout position of the widget immediately.

All widgets share a common set of properties, and each widget has its own specific properties. These properties are arranged in the following order on the Attributes tab of the Properties dialog:

- The Name attribute
- An alphabetical list of common and widget-specific properties, combined

On the Events tab of the Properties dialog, the properties are displayed in alphabetical order with common and widget-specific events combined.

For information on the properties you can set for each widget, see "Widget Properties" on page 647.

Entering Multiple Strings for a Property

There are several widget properties that you can set to multiple string values. The attribute's Value field contains a popup edit control in which you can enter multiple strings.

To enter more than one string in the edit control, do one of the following:

- Type in a string, then press Control+Enter at the end of each line.
- Type in a string, then press Control+j at the end of each line.

These actions move you to the next line. When you have entered the necessary string, press Enter to set the values.

Using the Widget Browser

The Widget Browser of the IDL GUIBuilder is a dialog window that presents the current GUI in a tree control. This presents the user with a different view into the GUI they are designing.

To start the Widget Browser, right-click on any component in an IDL GUIBuilder window, then choose **Browse** from the menu. This opens the Widget Browser, like the one shown in the following figure.

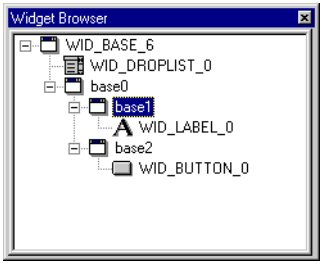


Figure 24-8: Widget Browser

The Widget Browser is helpful when you want to see your widget hierarchy and when you need to operate on overlapping widgets in your interface layout, which can happen when you design an interface to show or hide widgets on specific events. For an example that uses the Widget Browser for this purpose, see "Controlling Widget Display" on page 642.

Note

In the Widget Browser, there is no indication of defined menus.

You can expand the widget tree by clicking on the plus sign, or collapse it by clicking on the minus sign.

When you select a widget in the hierarchy by clicking on it, the widget is selected in the IDL GUIBuilder window, and the Properties dialog updates to display the selected widget's properties.

Right-click on a component to display a context menu from which you can cut, copy, paste, or delete the widget. From the context menu, you can also open the Properties dialog and the Menu Editor, when appropriate. To delete a widget from the Widget Browser, use the context menu, or select a widget and press the Delete key.

To change a widget's Name attribute in the Widget Browser, select the widget name with two single clicks on the name. This changes the name into an editable text box in which you can enter the new name. The Name attribute must be unique to the widget hierarchy.

For more information on other ways to operate on widgets, see "Widget Operations" on page 628.

Using the Menu Editor

You can add menus to top-level bases or to buttons that have the Type attribute set to Menu. To define menus for your interface, use the Menu Editor, which is shown in the following figure with defined menus. This dialog allows you to define menus, menu items, submenu titles, and submenus, and all their associated event procedures.

For instructions on how to define the menus shown in the following figure, see "Defining Menus for the Top-level Base" on page 602.

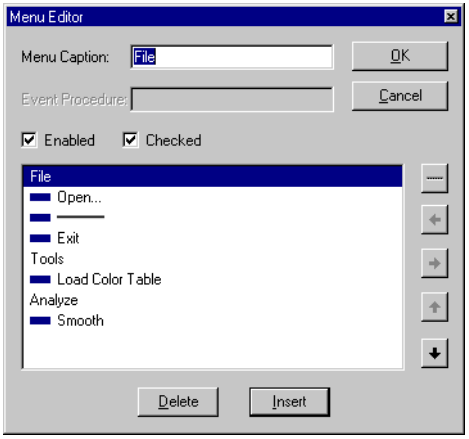


Figure 24-9: Menu Editor Dialog

Adding Menus to Top-Level Bases

To define basic menus, menu items, submenu titles, submenus, and their associated event procedures to top-level bases, follow these general steps:

1. Open the Menu Editor by doing one of the following:
   - Select the top-level base and select **Edit → Menu** from the IDLDE menu.
   - Right-click on the top-level base, then choose **Edit Menu**.
2. To define a top-level menu in the Menu Editor, enter a Menu Caption, and click Insert. When you are defining menus for a top-level base, the top-level menus are aligned along the left edge of the menu list, and the indentation indicates the nesting in the menu.

Note

The Menu Caption is the name that appears on the menubar. If you are defining a top-level menu for a base, you do not need to supply a value in the Event Procedure field. On button menus, however, where the button's Label attribute acts as the top-level menu, the first level of menus in the editor serve as menu items, and thus require a value in the Event Procedure field.

3. To define a menu item on a new line in the editor, click the right arrow, enter a Menu Caption and its associated event procedure, then click Insert. The Menu Caption is the name you want to appear on the menu. The Event Procedure is the name of the routine that will be called when the menu item is selected.

Note

For top-level bases, you must indent a line to make it a menu item and enable the Event Procedure field.

4. To define a submenu title, enter the Menu Caption, and click Insert. It is not necessary to define an Event Procedure for a submenu title.
5. To define submenus to a submenu title, enter the Menu Caption and the Event Procedure, indent the item another level by using the right arrow, and click Insert. Enter the submenus you want at this level of indentation.
6. To define another top-level menu or menu item, enter the information, click the left arrow until the indentation is appropriate, and click Insert.
7. To define a separator, select a blank line, or select the line you want the separator after, then click the separator button (which has a line on it and is above the arrow buttons).
8. To save your defined menus, Click OK in the Menu Editor. When you do so, the menu items will appear on the top-level base. To test the display of the menus, click on them.

Note

Under Microsoft Windows, including the ampersand character (&) in the Menu Caption causes the window manager to underline the character following the ampersand, which is the keyboard accelerator. This functionality is supported in the Menu Editor. If you are designing an application to run on other platforms, however, avoid the use of the ampersand in the Menu Caption.

- To move a menu item to a new position: Select the menu item, click the up or down arrow on the right side of the dialog until the menu item is in the desired position, then click OK.
- To add a menu item in the middle of existing menu items: Select the line you want the new item to follow, then click Insert. This adds a new line, for which you can enter a Menu Caption and Event Procedure.
- To make a menu item display disabled initially: Click the Enabled checkbox (to uncheck it). All menu items are enabled by default.
- To enable the ability to place a check or selection box next to the menu item: Click the Checked checkbox. (Checkmarks are placed next to menu items via the SET_BUTTON keyword to WIDGET_CONTROL in the event handling routine.)
- To delete a menu item: Select the item, then click Delete.
- To delete a menu: Delete each contained menu item, then delete the top-level menu.

Adding Menus to Buttons

You can also create buttons that contain menus. To add a menu to a button, follow these basic steps:

1. Click on the Button widget tool on the toolbar, then click on the top-level base area. This creates a button of the default size.
2. Right-click on the button and choose **Properties** to open the Properties dialog.
3. In the Properties dialog, change the value of the Type attribute to Menu.
4. Right-click on the button, then choose **Edit Menu** to open the Menu Editor. You can define the menu items and submenus with the Menu Editor, using the general steps described in “Using the Menu Editor” on page 621.

Note

For buttons, the Label attribute acts as the top-level menu, and the first level of menus in the Menu Editor serve as menu items. Therefore, the first level requires a value in the Event Procedures field (unlike top-level menu items for bases).

5. After you have defined all the necessary menus, click OK. When you do so, the menus are saved, and the button Label attribute is displayed as the top-level menu.

To view menus on buttons, do one of the following:

- Immediately after creating the menu (after clicking OK in the Menu Editor), click on the button, and the button menus will be displayed.
- At any other time, right-click on the button, and then choose **Show Menu**.

Using the Bitmap Editor

Use the Bitmap Editor to create 16 color bitmaps to be displayed on push buttons. The Bitmap Editor can read and write bitmap files (`*.bmp`). Using the editor, you can create your own bitmaps, or you can open existing bitmap files and modify them.

IDL supplies a set of bitmap files you can use in the buttons of your applications. The files are always available for loading. The bitmaps are located in the following directory:

```
IDL_DIR\resource\bitmaps
```

Placing a Color Bitmap on a Button

To display a bitmap on a button, follow these steps:

1. Right-click on the button widget, and choose **Properties** from the menu, which opens the Properties dialog for this button.
2. In the Type field, select Bitmap from the droplist.
3. In the Properties dialog, click on the arrow to the right of the Bitmap attribute, and do one of the following:
   - To place an existing bitmap on the button: Choose Select Bitmap, and select a bitmap file from the Open dialog. Note that when Bitmap type is selected, the Label attribute value changes to Bitmap.
   - To edit an existing bitmap and place it on the button: Choose Edit Bitmap, then select the bitmap file from the Open dialog. This opens the bitmap in the Bitmap Editor. The bitmap is displayed on the button when you save the file.
   - To create a new bitmap and place it on a button: Choose New Bitmap. This opens the Bitmap Editor, which you can use to create the new bitmap. When you save the `*.bmp` file, it is placed on the button.

   When you complete one of these processes, the filename of the selected bitmap appears in the Bitmap field of the Properties dialog, and the bitmap is displayed on the button.

Note

For 16- and 256-color bitmaps, IDL uses the color of the pixel in the lower left corner as the transparent color. All pixels of this color become transparent, allowing the button color to show through. This allows you to use bitmaps that are not rectangular. If you have a rectangular bitmap that you want to use as a button label, you must either draw a border of a different color around the bitmap or save the bitmap as 24-bit (TrueColor). If your bitmap also contains text, make sure the border you draw is a different color than the text, otherwise the text color will become transparent.

Using the Bitmap Editor Tools

The Bitmap Editor tools allow you to select from the color palette, and then use the Pencil (pixel fill), the Flood fill (fill clear area), or the Eraser (clear or color areas). The Bitmap Editor tools are shown in the following figure.

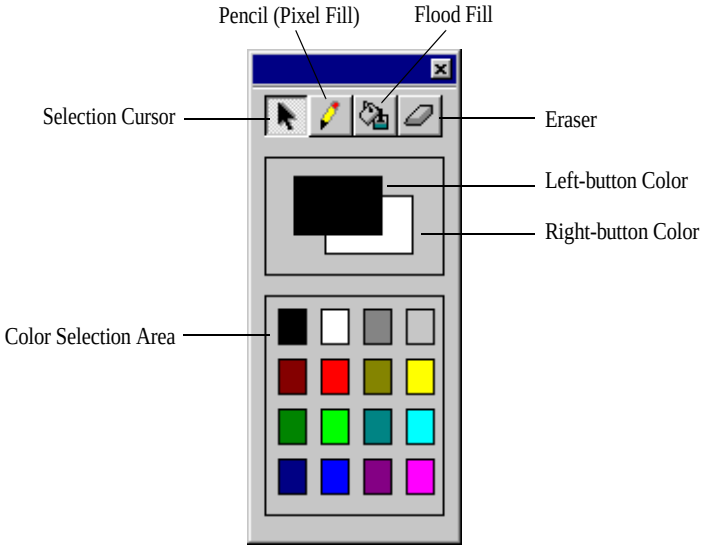


Figure 24-10: Bitmap Editor Tools

You can select a color by clicking on it in the color selection tool, or you can select your primary colors, the left-button and right-button colors, and then click on a tool and draw on the bitmap canvas. You can change the primary color selections at any time.

- To select the left mouse button color: Left-click on the color in the color selection area.
- To select a right mouse button color: Right-click on the color in the color selection area.
- To use the left color: With a tool selected, click or press and drag the right mouse button on the bitmap canvas.
- To use the right color: With a tool selected, click or press and drag the left mouse button on the bitmap canvas.
- To change the size of the bitmap: Drag the bitmap canvas to the desired size.

Using the Tree Editor

To define a tree widget hierarchy for your interface, use the Tree Editor, shown in the following figure. This dialog allows you to define tree nodes and folders, menus, menu items, submenu titles, and submenus, and all their associated event procedures.

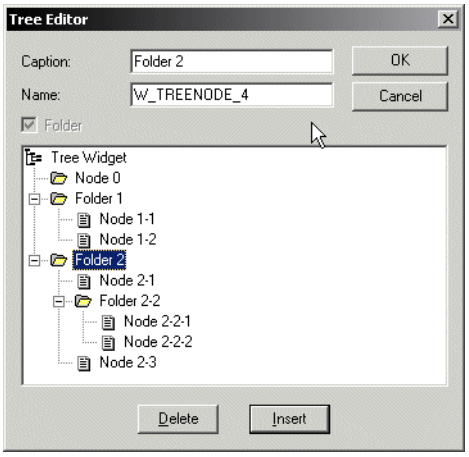


Figure 24-11: Tree Editor Dialog

To create a tree hierarchy, do the following:

1. Add a tree widget to your interface, right-click on the widget, and choose **Edit Tree**. A new tree widget is created, beginning with the root node of the tree.
2. Click **Insert** to add a node to the tree. Nodes are added as leaf nodes below whichever node is currently selected. Inserting a node below an existing leaf

node turns the existing leaf node into a branch node (that is, a *folder*). You can also turn a leaf node into a branch node by selecting the node and clicking the **Folder** checkbox.

3. Modify the **Caption** and **Name** fields for each node as desired. The value of the **Caption** field will be come the VALUE of the tree widget; this string is used as the text label when the widget is displayed. The value of the **Name** field will become the variable name of the variable that holds the tree widget's widget ID.
4. To delete a node from the hierarchy, select the node and click **Delete**. Any nodes or that have the selected node as their parent will also be deleted.
5. Expand and collapse nodes within the editor by clicking on the plus and minus symbols to the left of the branch nodes.
6. Rearrange the nodes and branches as desired by dragging and dropping the nodes' captions or icons.
7. Click **OK** to save your changes to the tree widget hierarchy, or **Cancel** to abandon your changes.

Widget Operations

The IDL GUIBuilder allows you to operate on widgets in many ways. You can select, deselect, move, cut, copy, paste, and delete widgets, and you can undo and redo operations. This section describes the following:

- Selecting Widgets
- Moving and Resizing Widgets
- Cutting, Copying, and Pasting Widgets
- Deleting Widgets
- Undoing and Redoing Operations

Selecting Widgets

You can select a widget, then move it or resize it.

To select a widget, click on the widget.

To select more than one widget, do one of the following:

- Press Shift and click on each widget.
- Press Control and click on each widget. When you press Control, you can change the selection state by clicking again on the widget; pressing Control during selection allows you to toggle the selection state of a widget without affecting the selection state of any other widget.
- Press the left mouse button and drag out an area in the top-level base that includes the widgets you want to select. When you release the mouse button, widgets in the selection box are selected.

When you select multiple widgets, there is always one primary selection. The primary widget selection is indicated with the dark, filled-in selection handles. If you open the Properties dialog with multiple widgets selected, the properties displayed are those for the primary selection.

Note

When selecting multiple widgets, you can select only widgets that share the same base widget as their parent.

Moving and Resizing Widgets

You can move widgets around in their parent base by dragging the widget to a new location or by using the arrow keys.

To move a widget to a new base, or to give a widget a new parent base within the same top-level base, do one of the following:

- Press Alt and drag and drop the widget on the new parent base.
- Right-click on the widget, choose Cut from the menu, right-click on the new base widget, and choose Paste from the menu.

To resize a widget, click on a sizing handle, and drag to the desired size. To size the widget larger than its parent base, press Alt and drag to the desired size.

Cutting, Copying, and Pasting Widgets

You can cut, copy, and paste widgets within the same base or to another base in another IDL GUIBuilder window, using the Edit menu items, toolbar buttons, or a context menu (opened with a right-click on the widget).

To cut or copy a selected widget, or to paste a widget from the clipboard, do one of the following:

- Choose the desired operation from the **Edit** menu, or from the IDLDE toolbar.
- Right-click on the widget and select the desired operation from the menu. If you are pasting, right-click on the base widget you want to paste into.
- Select the widget and use standard windows keyboard shortcuts to cut, copy, or paste the widget.

Note

All cut or copied items are placed on a local clipboard, not on the system clipboard.

Deleting Widgets

To delete a widget, do one of the following:

- Select the widget and choose **Edit** → **Delete**.
- Select the widget and press the Delete key.
- Right click on a widget and choose **Delete** from the menu.

Undoing and Redoing Operations

In the IDL GUIBuilder, you can undo or redo unlimited operations between save procedures. If you save the resource file, the operations are cleared from memory.

To undo an operation, do one of the following:

- Select **Edit → Undo**.
- Click the “Undo” button on the IDLDE toolbar.
- Press Control+z.

To redo an operation, do one of the following:

- Select **Edit → Redo**.
- Click the “Redo” button on the IDLDE toolbar.
- Press Control+y.

Generating Files

The IDL GUIBuilder generates the following two types of files:

- `*.prc` files that contain the resource definitions for the interface definition as displayed in the IDL GUIBuilder.
- `*.pro` files that contain the generated IDL source code. The generated `*.pro` files are portable across all IDL-supported platforms.

Generating Resource Files

The `*.prc` files contain the resource definitions for the graphical interface. You can open `*.prc` files in the IDL GUIBuilder and modify the interface at anytime. Do not attempt to modify this file directly.

To save a `*.prc` file for the first time, choose **Save** or **Save As** from the IDLDE **File** menu. This opens the "Save As" dialog, which allows you to select a location and indicate a file name for the `*.prc` file.

To have the `.prc` file generate code for a project, open the `.prc` file and do the following for your platform:

- Windows: select **File** → **Generate**.
- UNIX: select **Project** → **Build**.

Generating IDL Code

The IDL GUIBuilder can generate these two kinds of `*.pro` IDL source code files:

- Widget definition code (`*.pro` files).
- Event-handling code (`*_eventcb.pro` files).

To save both the widget code and the event handler `*.pro` files, select **File** → **Generate .pro** from the IDLDE menu. This opens the "Save As" dialog, which you can use to select a location and indicate a name for the widget code. The event code file name is based on the name specified for the widget code. For example, if you enter `app1.pro` in the File name field, the event code file will be named `app1_eventcb.pro`.

Note

Never modify the generated `*.pro` interface file. If you decide to modify the application interface, use the IDL GUIBuilder, then regenerate the file. When you regenerate the widget code, the file is overwritten.

Note

When you save both files, IDL puts the RESOLVE_ROUTINE procedure in the generated widget code. The procedure contains the name of the related `*_eventcb.pro` event-handler file so that it will be compiled and loaded with when you run the widget code.

Notes on Generating Code a Second Time

When you modify an interface and save the `*.prc` file, it is overwritten, which should not be a problem. If you decide to change your interface, however, you will need to regenerate the widget code and thus overwrite the `*.pro` widget code file.

Note that if you regenerate either of the `*.pro` files, they are overwritten. When writing code, you should modify only the generated event-handling file (`*_eventcb.pro`). You should never modify the generated widget code (the `*.pro` file). This allows you to change the interface and regenerate the definition code without losing modifications in that file. This should simplify the procedures you need to take to update or change an interface.

Because it is modular, the event-handler code is simple to modify after you change the interface definitions. When you regenerate the IDL source code files, any new event handler code is appended to the end of the file.

IDL GUIBuilder Examples

After you define your interface and generate IDL code using the IDL GUIBuilder, you will write the code that controls the application's behavior. You can modify the code, compile it, and run it using the IDLDE.

Generally, you will be writing the event-handler callbacks for the procedures located in the generated `*_eventcb.pro` file. While doing this, you might like to handle initialization states, have multiple GUIs work together, add compound widgets, or control widget display. For examples of how to handle these different types of events, see the following sections:

- Understanding IDL GUIBuilder Event Handling Code
- Writing Event Callback Routines
- Handling Initialization Arguments
- Integrating Multiple Interfaces
- Adding Compound Widgets
- Controlling Widget Display

Understanding IDL GUIBuilder Event Handling Code

When using the IDL GUIBuilder, you assign event procedures to specific events using the Events tab of the Properties dialog. The calling sequence for the events that you set are added to the generated `*_eventcb.pro` event callback code.

The argument that is passed into the specified event routine depends on the type of event being processed. Creation, realization, and destruction event routines are usually passed the ID of the involved widget, and all other callback routines are passed the appropriate IDL widget event structure.

It is a normal operation in applications to change the attributes of the interface when certain events occur. One method used in handling events for IDL GUIBuilder generated applications is the UNAME keyword, or the Name attribute, given to all created widgets. (In a programmatically-created IDL application, this action is handled using information stored in a widget component's user value.)

When you create a widget in the IDL GUIBuilder, IDL gives it a name unique to the widget hierarchy to which it belongs. You can rename the widget using the Name attribute.

In the generated code, this name is specified by the UNAME keyword. Because these names are unique, you can use the WIDGET_INFO function with the FIND_BY_UNAME keyword in your event callback routines to get the IDs of widgets in the interface application.

Note

For information on properties, see "Using the Properties Dialog" on page 617, and see "Widget Properties" on page 647.

Writing Event Callback Routines

This short example shows how basic event processing works in code generated by the IDL GUIBuilder. The example demonstrates how to use the FIND_BY_UNAME keyword to obtain the IDs of other widgets in the interface.

To create this simple example application, follow these steps:

1. Select **File** → **New** → **GUI** from the IDLDE menu. This opens a new IDL GUIBuilder window.
2. In the IDL GUIBuilder window, right-click on the contained top-level base, and choose **Properties** from the menu. This opens the Properties dialog.
3. In the open Properties dialog, click the push pin button to keep the dialog open and on top.
4. On the Attributes tab of the Properties dialog, set the top-level base Layout attribute to Column.
5. On the IDL GUIBuilder toolbar, click the Label Widget button, and click on the top-level base area to add a label widget to the base.
6. With the label widget selected, set the following attributes in the Properties dialog:
   - In the Name field, enter "clock".
   - Set the Alignment attribute to Center.
   - Set the Component Sizing attribute to Default.
   - In the Text field, enter "No Time Currently Available".
7. On the IDL GUIBuilder toolbar, click the Button Widget button.
8. Click on the top-level base area, which adds a button widget to the interface.
9. With the button selected, set the Label attribute to "Time".

10. In the Properties dialog, click the Events tab and set OnButtonPress to “OnPress”.

Your interface definition should look like the one shown in the following figure.

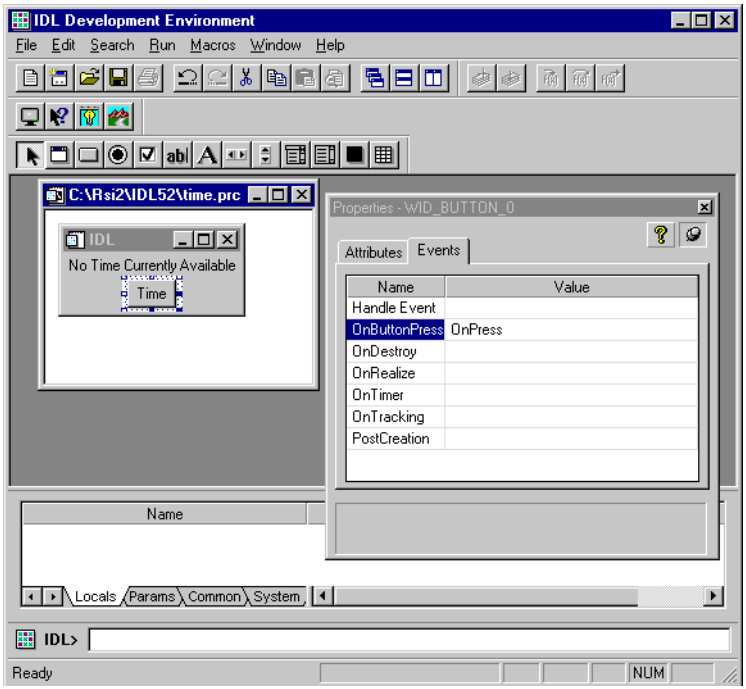


Figure 24-12: Handling Events Example Application

11. Select **File** → **Save** from the IDLDE menu, which opens the “Save As” dialog.
12. In the “Save As” dialog, select a location, enter “time.prc” in the File name field, and click Save. This saves the interface definition to a resource file.
13. Select **File** → **Generate .pro** from the IDLDE menu. In the “Save As” dialog, select the location, enter “time.pro” in the File name field, and click Save. This saves the `time.pro` widget code file and the `time_eventcb.pro` event callback code to the specified directory.
14. Select **File** → **Open** from the IDLDE menu. In the “Open” dialog, select the `time_eventcb.pro` file and click Open.
15. In the `time_eventcb.pro` file, locate the OnPress event procedure place holder, which looks like this:

```
pro OnPress, Event

end
```

16. Add the following IDL code between the PRO and END statements to handle a button press:

    ```
    ; Get the widget ID of the label widget.
    Label = widget_info(Event.top, find_by_uname='clock')

    ; Set the value of the label widget to current time.
    widget_control, Label, set_value=Systime(0)
    ```

    The first command gets the ID of the label widget by searching the widget hierarchy for a widget named “clock”. This is the name that you gave the label widget in the IDL GUIBuilder Properties dialog. Once the ID is found, the second command sets the value of the label widget to the current system time.

17. Select **Run → Compile time_eventcb.pro** to save and compile the file.

18. To execute the program, enter `time` at the IDL command prompt.

This compiles and runs the `time.pro` file. In the running application, you can press the Time button to cause the current time to be displayed in the label.

Handling Initialization Arguments

You can provide runtime initialization information to the generated `*.pro` widget code by modifying the `*_eventcb.pro` file. Keywords provided to the generated widget interface procedure are passed to the post creation routines using the _EXTRA keyword.

If a routine is defined with the _EXTRA keyword parameter, you can add unrecognized keyword and value pairs, and the pairs are placed in an anonymous structure. The name of each unrecognized keyword becomes a tag name, and each value becomes the tag value.

You will use this feature most often when your application launches floating or modal dialogs, but the functionality is always available.

For example, if you want to display a dialog at the creation of an application, you would follow these basic steps:

1. Create an interface using the IDL GUIBuilder.

2. After creating the interface, open the Properties dialog for the top-level base and set the PostCreation event for the top-level base widget to a routine name, such as “OnCreate”.

3. Save the interface definition and generate the IDL source code.
4. In the generated `*_eventcb.pro` event code file, locate the “OnCreate” routine place holder, which looks like this:

```
pro OnCreate, wWidget, _EXTRA=_VWBExtra_

end
```

5. To process a specific keyword in this post creation routine, declare the keyword in the procedure statement and add the processing code to the procedure.

   For example, to process the DO_DIALOG keyword in the defined OnCreate procedure, add the DO_DIALOG keyword to the procedure, and add the logic to handle it to the event callback routine. The completed procedure should look like this:

```
pro OnCreate, wWidget, DO_DIALOG=DO_DIALOG, _EXTRA=_VWBExtra_

   ; If DO_DIALOG is set, display a simple message box.
   if( Keyword_Set(DO_DIALOG) )then $
     status = Dialog_Message("On Dialog Set")

end
```

6. Save the file, then compile and generate the application. To show the dialog at creation time, enter the following at the IDL command prompt:

```
<ProgramName>, /DO_DIALOG
```

Integrating Multiple Interfaces

You can create multiple interfaces with the IDL GUIBuilder then integrate them to form the complete application hierarchy. This example shows you how to construct two interfaces and integrate them.

The first interface you will create is the main window, and it will consist of a simple push button that will launch a modal dialog. The second interface you will create is the modal dialog, and it will display a close button.

Creating the Main Window

To create the main window, follow these steps:

1. Select **File** → **New** → **GUI** from the IDLDE menu to open a new IDL GUIBuilder window with a top-level base.

2. On the IDL GUIBuilder toolbar, click on the Button Widget button, then click on the top-level base. This adds a button of the default size to the base. You can place the button anywhere in the base.
3. Right-click on the newly created button, and choose **Properties** from the context menu to open the Properties dialog.
4. In the Properties dialog, click the push pin button to keep the dialog open and on top.
5. Set the button's Label attribute to "Modal Dialog".
6. Click on the Properties dialog Events tab, and set the OnButtonPress value to "OnPress".
7. Select **File** → **Save**. In the "Save As" dialog, select a location, enter "maingui.prc" in the File name field, and click Save. This saves the interface definition to an IDL resource file.
8. Select **File** → **Generate .pro**. In the "Save As" dialog, select a location, enter "maingui.pro" in the File name field, and click Save. This saves the `maingui.pro` widget code and the `maingui_evnetcb.pro` event-handler code.
9. Select **File** → **Open**. In the "Open" dialog, select the `maingui_eventcb.pro` file, and click Open.
10. In the `maingui_eventcb.pro` file, locate the OnPress event procedure place holder, which looks like this:

    ```
    pro OnPress, Event

    end
    ```

11. Add the following code between the PRO and END statements:

    ```
    modalgui, group_leader=Event.top
    ```

    You will create the "modalgui" dialog in the next set of steps. Note that you set the GROUP_LEADER keyword here because the modal dialog requires it.
12. Select **Run** → **Compile maingui_eventcb.pro**. This saves and compiles the file.

Creating the Modal Dialog

To create the modal dialog, follow these steps:

1. Open a new IDL GUIBuilder window.

2. In the IDL GUIBuilder window, select the top-level base, and set the following in the Properties dialog:
   - Set the Modal attribute to True.
   - In the Title field, enter “Modal Dialog”.
3. On the IDL GUIBuilder toolbar, click the button widget, then click on the top-level base. This adds a button to the top-level base. Place it anywhere in the base.
4. With the new button selected, set the Label attribute value to “OK”.
5. On the Events tab of the Properties dialog, set the OnButtonPress value to “OnModalPress”.
6. Select **File → Save**. In the “Save As” dialog, select a location, enter “modalgui.prc” in the File name field, and click Save. This saves the interface definition to an IDL resource file.
7. Select **File → Generate .pro**. In the “Save As” dialog, select a location, enter “modalgui.pro” in the File name field, and click Save. This saves the `modalgui.pro` widget code file and the `modalgui_eventcb.pro` event callback file.
8. Open the `modalgui_eventcb.pro` file and locate the OnModalPress procedure place holder. Then add the following code between the PRO and END statements so that the dialog closes when the button is pushed:

   ```
   widget_control, Event.top, /destroy
   ```
9. Save and compile this file.

Running the Example Application

Enter `maingui` at the IDL command prompt. This command runs the main window. You can press the Modal Dialog button, and the modal dialog is displayed. When you press the OK button on the modal dialog, the dialog exits.

Adding Compound Widgets

The IDL GUIBuilder tools do not allow you to add a compound widget directly to your interface. You can, however, modify your event code to add a compound widget.

To add a compound widget to an IDL GUIBuilder generated interface, follow these basic steps:

1. Add the compound widget to the widget tree in a PostCreation event callback procedure.
2. Handle the events generated by the compound widget in the Handle Event callback function. Set this event function value for the base widget that will contain the compound widget.

Adding a Compound Widget to an Interface

This example demonstrates how to add a compound widget to an application constructed with the IDL GUIBuilder. The application contains a label and a CW_FSLIDER compound widget. In the running application, the values generated by CW_FSLIDER will be displayed in the label widget.

To create this application, follow these steps:

1. Select **File** → **New** → **GUI** from the IDLDE menu to open a new IDL GUIBuilder window with a top-level base.
2. Right-click on the base and choose **Properties** to open the Properties dialog for the top-level base.
3. In the Properties dialog, click the push pin button to keep the dialog on top.
4. In the Properties dialog of the top-level base, set the Layout attribute to Column.
5. To add the label, click the Label Widget button on the toolbar, then click on the top-level base. This creates a label widget of the default size.
6. With the label selected, set the following in the Properties dialog:
   - In the Name value field, enter “label”.
   - Set the Alignment attribute to Center.
   - Set the Component Sizing attribute to Default.
   - In the Text value field, enter “000.000”.
7. Click the Base Widget button on the toolbar, and click on the top-level base. This adds a base to the top-level base.
8. With the new base widget selected, set the Component Sizing attribute to Default.
9. In the Properties dialog, click on the Events tab and set the following base widget event values:

- In the Handle Event Value field, enter “HandleEvent”. This is the name of the function that will handle the compound widget events.
- In the PostCreation Value field, enter “AddCW”. This is the name of the event routine that will create the compound widget.

10. Select **File** → **Save**. In the “Save As” dialog, select a location, enter “compound.prc” in the File name field, and click Save. This saves the interface definition to an IDL resource file.
11. Select **File** → **Generate .pro**. In the “Save As” dialog, enter “compound.pro”, and click Save. This generates the `compound.pro` widget code file and the `compound_eventcb.pro` event-handler file.
12. Select **File** → **Open**, and open the `compound_eventcb.pro` file.
13. In the `compound_eventcb.pro` file, locate the AddCW event routine place holder, and insert the code to add the CW_FSLIDER compound widget to the base widget. The routine should look like this:

```
pro AddCw, wWidget

   idslide = CW_FSLIDER(wWidget, /SUPPRESS_VALUE)

end
```

14. Add the event callback routines to the generated HandleEvent function. The function should look like this:

```
FUNCTION HandleEvent, Event

   ; Fslider event structure is an anonymous structure, so
   ; the following will return "" if it is from fslider.

   IF(TAG_NAMES(Event, /STRUCTURE_NAME) eq "")THEN BEGIN

      ; Get the id of the label widget using its name.
      id = widget_info(Event.top, find_by_uname='label')

      ; Set the value of the label, to the value in the slider.
      WIDGET_CONTROL, id, set_value= $
         String(Event.value, format='(f5.2)')
      RETURN,0
      ; Halt event processing here.
   ENDIF

   RETURN, Event

END
```

Note that the callback routine finds the label widget using the FIND_BY_UNAME keyword with the name value you gave the widget in the Properties dialog.

15. Select **Run → Compile compound_eventcb.pro** to save and compile the file.

Running the Example

To run the application, enter `compound` at the IDL command prompt. This complies and runs the application. In the running application, move the CW_FSLIDER and the value is placed in the label.

Controlling Widget Display

This example demonstrates how to use the IDL GUIBuilder to create an interface that contains overlapping sub-bases containing different types of widgets. The example shows how you can display and hide overlapping controls in an interface created in the IDL GUIBuilder, and it incorporates using the Widget Browser. Note that this example is slightly more complicated than the others.

This example constructs an interface with the following widgets:

- A droplist.
- A sub-base that contains two sub-bases:
  - One sub-base containing a text widget.
  - One sub-base containing a button.

The two contained sub-bases overlap and the visibility of each is controlled by the value selected in the droplist. When users select an item in the droplist, one sub-base is hidden and the other one is displayed.

Creating the Interface

To create this application interface, follow these steps:

1. Select **New → GUI** from the IDLDE File menu to open a new IDL GUIBuilder window with a top-level base.
2. Right-click on the top-level base, and choose **Properties** from the menu. This opens the Properties dialog.
3. In the Properties dialog, click the push pin button to keep the dialog open and on top.
4. In the Properties dialog, set the Layout attribute to Column.

5. On the IDL GUIBuilder toolbar, click on the Droplist Widget button, then click on the top-level base. This creates a droplist in the base area.
6. With the droplist selected, set the following in the Properties dialog:
   - In the Title value field, enter “Active Base”.
   - In the Initial Value field, click on the arrow. This displays a popup edit control. Enter “Base One”, press Control+Enter to move to the next line, enter “Base Two”, and press Enter to close the popup edit control.
7. On the Events tab of the Properties dialog, set OnSelectValue to “OnSelect”.
8. On the IDL GUIBuilder toolbar, click on the Base Widget button, then click on the top-level base. This adds a base widget of the default size to the interface.
9. With the new base selected, set the following attributes in the Properties dialog:
   - In the Name value field, enter “base0”.
   - Set the Frame attribute to True.
10. On the IDL GUIBuilder toolbar, click on the Base Widget button, then click on the base you just added. This adds a base widget to the “base0” widget.
11. With the newly-added base selected, set the following attributes in the Properties dialog:
    - In the Name value field, enter “base1”.
    - Set the Component Sizing attribute to Explicit.
    - In the X Offset value field, enter “0”.
    - In the X Size value field, enter “200”.
    - In the Y Offset value field, enter “0”.
    - In the Y Size value field, enter “200”.
12. Right-click on a base, and choose **Browse** from the context menu. This opens the Widget Browser.
13. In the Widget Browser, right-click on base1, and choose **Copy**, which copies the widget to the local clipboard.
14. In the Widget Browser, right-click on “base0”, and choose **Paste**, which pastes the copied base in to the “base0” widget. The new base is called “base1_0”.

15. In the Widget Browser, select “base1_0”. This selects the base in the IDL GUIBuilder window and updates the Properties dialog with the appropriate properties and values.
16. With “base1_0” selected, set the following attributes in the Properties dialog:
    - In the Name value field, enter “base2”.
    - Set the Component Sizing attribute to Explicit.
    - In the X Offset value field, enter “0”.
    - In the X Size value field, enter “200”.
    - In the Y Offset value field, enter “0”.
    - In the Y Size value field, enter “200”.
17. Select **File** → **Save**. In the “Save As dialog”, select a location, enter “visible.prc” in the File name field, and click Save. This saves the interface definition.
18. In the Widget Browser, select “base1”.
19. With “base1” selected, set the Visible attribute to False. This will hide “base1” and make “base2” visible.
20. On the IDL GUIBuilder toolbar, click the Button Widget button, then click on “base2” in the IDL GUIBuilder. This adds a button to the base widget. Place the button anywhere in this base.
21. With the button selected, set the Label attribute to “Button 2”.
22. In the Widget Browser, select “base2”, and using the Properties dialog, set the Visible attribute to False to hide the base.
23. In the Widget Browser, select “base1”, and set the Visible attribute to True to show the base.
24. On the IDL GUIBuilder toolbar, click the Label Widget button, then click on “base1”. This adds a label to “base1”. Place the label anywhere in this base.
25. With the label widget selected, set the Text attribute to “Label 1”.
26. Select **File** → **Save** to save the changes to the `visible.prc` resource file.

The interface is now complete. It should look similar to the one shown in the following figure.

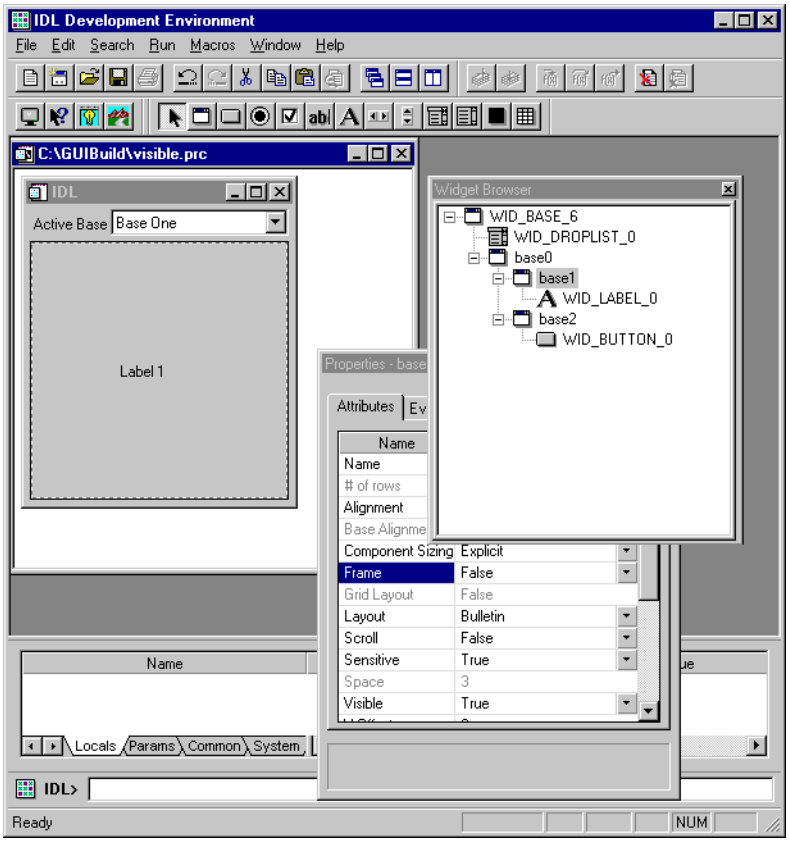


Figure 24-13: Visible Widgets Example Application

Generating and Modifying the Code

To generate and modify the code, follow these steps:

1. Select **File** → **Generate .pro**. In the “Save As” dialog, select a location, enter “visible.pro” in the File name field, and click Save. This saves the `visible.pro` widget code file and the `visible_eventcb.pro` event-handler file.
2. Select **File** → **Open**, select the `visible_eventcb.pro` file, and click Open.

3. In the `visible_eventcb.pro` file, locate the OnSelect event procedure place holder, which looks like this:

```
pro OnSelect, Event

end
```

4. Add the following code between the PRO and END statements:

```
; Toggle the mapping of the two IDL sub-bases and
; get the Widget IDs of the two sub-bases.
wBase1 = Widget_Info(Event.top, find_by_uname="base1")
wBase2 = Widget_Info(Event.top, find_by_uname="base2")

; Now update the mapping.
widget_control, wBase1, map=(Event.index eq 0)
widget_control, wBase2, map=(Event.index eq 1)
```

   The added IDL code gets the Widget IDs of the sub-bases that you created and sets the mapping (hide or show) of these bases depending on the selected value of the droplist.

5. Select **Run → Compile visible_eventcb.pro** to save and compile the file.

Running the Application

To run this application, enter `visible` at the IDL command prompt. This command executes the visible application. In the running application, you can change the selection in the droplist, and the action will change the displayed widget.

Widget Properties

For each widget type, there is a set of attribute values and a set of event values you can set using the IDL GUIBuilder Properties dialog. When you select a widget in the IDL GUIBuilder window or in the Widget Browser, the Properties dialog is updated to contain the properties for the selected widget. These properties include those common to all widgets and those specific to the selected widget.

On the Attributes tab of the Properties dialog, the properties are set to default values and are arranged in the following order:

- The Name attribute.
- An alphabetical list of common and widget-specific properties, combined.

On the Events tab, the possible events for a widget are listed in alphabetical order, with the common and the widget-specific events combined. By default, no event values are set initially. When you enter a routine name for an event property, you are responsible for making sure that event procedure exists. IDL does not validate the existence of the specified routine.

For information on how to open and use the Properties dialog, see "Using the Properties Dialog" on page 617.

The rest of this chapter describes the properties you can set for each widget:

Common Widget Properties

There are several attribute and event property values you can set for all widgets. The attribute properties include the name of the widget and the sizing properties. The event properties include creation, realization, destruction, and tracking events.

The following sections describe the common properties:

- Common Attributes
- Common Events

Common Attributes

The following attributes are common to all widgets:

Name

The Name attribute specifies the name of the component. This value can be any string that is unique to the widget hierarchy of the interface, but the string cannot contain spaces. For each widget you create in the IDL GUIBuilder, a default name is supplied, and this name is in the WID_<*TYPE*>_<*NUMBER*> format.

If you copy and paste a widget in the IDL GUIBuilder, the new widget is given a unique name based on the name of the one you copied. A number is added to the first widget's name, or an existing number is incremented.

You can use the Name value for the widget in your event callback routines. For example, you can use the specified name to find the widget, using the FIND_BY_UNAME keyword to the WIDGET_INFO function. Set the name for each widget to a name that makes sense to you; set the name value to something that is easy to remember and easy to use in your code.

In the generated `*.pro` file, this value is specified with the UNAME keyword to the widget creation routines.

Component Sizing

The Component Sizing keyword determines how the component is sized, which is by one of the following methods:

- Default: The widget is sized to a natural or implicit size. This is the default setting for the attribute. For example, a label widget's natural size is determined by the size of the text it is displaying with extra space for margins. The default size for each widgets is controlled by several things, including

displayed font size and the characteristics of the operating system displaying the interface.

- Explicit: The widget size is determined by several attributes, which include Layout for the base and its own X Size and Y Size keywords.

In the generated `*.pro` widget file, this value is specified with the XSIZE and YSIZE keywords to the widget creation routines.

Note

The default size of text widgets on Motif is based on the width of text, but the default size for text widgets on Windows is approximately 20 characters.

Frame

The Frame attribute determines if the widget will have a frame or border around it. These are the possible values:

- False: The widget will have no frame drawn around it. This is the default value.
- True: The widget will have a frame or border around it.

In the generated `*.pro` widget file, this value is specified by the FRAME keyword to the widget creation routines.

Note

The Frame attribute is not available for top-level base widgets.

Sensitive

The Sensitive attribute determines if the selected widget is active or not active on startup. You can set this value to determine if the user can access and manipulate the widget immediately after creation. These are the possible values:

- True: The widget is initially displayed as enabled and accepts keyboard or mouse input and generates events. This is the default value.
- False: The widget is initially displayed as disabled and does not accept keyboard or mouse input. The appearance of most widgets change when the False value is set, but the appearance does not always change to indicate this state.

In the generated `*.pro` file, this value is specified with the SENSITIVE keyword to the widget creation routines.

Note

To change the sensitivity of a widget after the widget is created, use the WIDGET_CONTROL function with the SENSITIVE keyword.

X Offset

The X Offset attribute specifies the X offset of the component from its parent. The possible values for X Offset are *o* to *n*, in pixels; any number is valid. The Y Offset attribute specifies the Y offset.

In the generated `*.pro` file, this value is specified with the XOFFSET keyword to the widget creation routines.

Note

The X Offset attribute value is *not* used with base widgets that have the Layout attribute set to Row or Column.

X Size

The X Size attribute specifies the width of the visible component in pixels. This attribute is disabled when Component Sizing is set to Default (and the default size is used). To enable this value, set Component Sizing to Explicit. The possible values for X Size are 0 to *n*, in pixels.

In the generated `*.pro` file, this value is specified with the SCR_XSIZE keyword to the widget creation routines.

Note

If you add scroll bars to a widget, use the widget-specific X Scroll attribute to set the width of the virtual area.

Y Offset

The Y Offset attribute specifies the Y offset of the component from its parent in pixels. The possible values for Y Offset are 0 to *n*, in pixels; any number is valid. The X Offset attribute specifies the X offset.

In the generated `*.pro` file, this value is specified by the YOFFSET keyword to the widget creation routines.

Note

The Y Offset attribute value is *not* used with base widgets that have the Layout attribute set to Row or Column.

Y Size

The Y Size attribute specifies the height of the visible component in pixels. This attribute is disabled when Component Sizing is set to Default (and the default size is used). To enable this value, set Component Sizing to Explicit. The possible values for Y Size are 0 to *n*, in pixels.

In the generated `*.pro` file, this value is specified with the SCR_YSIZE keyword to the widget creation routines.

Note

If you add scroll bars to a widget, use the widget-specific Y Scroll attribute to set the height of the virtual area.

Common Events

The following events are common to all widgets (by default, no event values are set):

Handle Event

The Handle Event value is the function name that is called when an event arrives from a widget that is rooted in an IDL GUIBuilder-created widget in the hierarchy. All events are sent to this event function, except for creation and destruction events.

For example, if you add a compound widget to an interface, using the PostCreation event procedure for a base widget, you should set the Handle Event value for that parent base widget (for the compound widget's parent widget). Then, you can handle all the events returned by the compound widget using this event function value.

In the generated `*_eventcb.pro` file, the event function place holder looks like this:

```
Function <Name>, Event

   return, Event
End
```

where *Name* is the name of the event function you specify. `Event` is the returned event structure, which is specific to the widget event.

For an example of how to handle the generated Handle Event function, see "Adding Compound Widgets" on page 639.

OnDestroy

The OnDestroy value is the routine name that is called when the widget is destroyed. In the generated `*_eventcb.pro` file, the event calling sequence looks like this:

```
pro <RoutineName>, wWidget
```

where *RoutineName* is the name of the event procedure you specify. wWidget is the IDL widget identifier.

OnRealize

The OnRealize value is the routine name that is called automatically when the widget is realized. In the generated `*_eventcb.pro` file, the event calling sequence looks like this:

```
pro <RoutineName>, wWidget
```

where *RoutineName* is the name of the event procedure you specify. wWidget is the IDL widget identifier.

OnTimer

The OnTimer value is the routine name that is called when a timer event is detected for a widget. In the generated `*_eventcb.pro` file, the event calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the returned event structure, which has the 3 standard event tags and looks like this:

```
{ WIDGET_TIMER, ID:0L, TOP:0L, HANDLER:0L }
```

You must set timer events for a widget, using the WIDGET_CONTROL function. The code generated by the IDL GUIBuilder only routes the events.

OnTracking

The OnTracking value is the routine name that is called when the widget receives a tracking event, which occurs when the mouse pointer *enters* or *leaves* the region of the widget. In the generated `*_eventcb.pro` file, the event calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the returned structure, which is of the following type:

```
{ WIDGET_TRACKING, ID:0L, TOP:0L, HANDLER:0L, ENTER:0 }
```

ENTER is 1 if the tracking event is an entry event, and 0 if it is an exit event.

PostCreation

The PostCreation value is the routine name that is called after the widget is created, but before it is realized. In the generated `*_eventcb.pro` file, the calling sequence looks like this:

```
pro <RoutineName>, wWidget
```

where *RoutineName* is the name of the event procedure you specify. wWidget is the IDL widget identifier.

Base Widget Properties

A base widget holds other widgets, including other base widgets. You can create groupings of widgets by using a base widget, thus forming a widget hierarchy.

When you open the IDL GUIBuilder, a top-level base is created, and you build your interface in this base. Top-level bases are a special class of the base widgets that are created without parent widgets; they act as the top-level parent in the widget hierarchy.

In the IDL GUIBuilder, you can add a menubar to the top-level base by using the Menu Editor.

In addition, you can make top-level bases *float* above their group leaders, with the Floating attribute, or you can make them *modal* dialogs, with the Modal attribute. Modal dialogs interrupt program execution until the user closes them. When you make a top-level base floating or modal, you must provide a group leader when calling the generated code, by using the GROUP_LEADER keyword.

When programming in IDL, you create base widgets using the WIDGET_BASE function. For more information, see "WIDGET_BASE" in the *IDL Reference Guide* manual.

For more information on the Menu Editor, see "Using the Menu Editor" on page 621.

Note

A base widget's layout is controlled by where you place it and the properties of its parent base.

Base Widget Attributes

For base widgets, you can set common attributes and base-specific attributes. For a list attributes common to all widgets, see "Common Attributes" on page 648.

Some of the base widget attributes apply to top-level bases only, and this limitation is noted in the following list of base widget attributes:

of Rows/Columns

The # of Rows/Columns attribute specifies the number of Columns or Rows to use when laying out the base. This attribute is valid only when the Layout attribute is set to Column or Row. The possible values for this setting are 1 to *n*, and the default value is 1.

In the generated `*.pro` file, this value is specified with the COLUMN or the ROW keyword to the widget creation routine.

For information on other properties that control the layout of contained widgets, see Alignment, Layout, Space, X Pad, and Y Pad.

Alignment

The Alignment attribute defines how components are aligned in the base. The way in which the value of this attribute affects the display of widgets depends on the value of the Layout attribute. The following is a list possible values for the Alignment attribute, and each value description includes information on how it works with the Layout settings:

- Center: Aligns the contained widgets with the center this parent base. This is the default value. For this setting to take effect, the Layout setting must be Row or Column. With Row set, the contained widgets are vertically centered. With Column set, the contained widgets are horizontally centered.
- Top: Aligns contained widgets with the top of this parent base. For this setting to take effect, the Layout setting must be Row.
- Bottom: Aligns the contained widgets with the bottom of this parent base. For this setting to take effect, the Layout setting must be Row.
- Left: Aligns the contained widgets with the left side of this parent base. For this setting to take effect, the Layout setting must be Column.
- Right: Aligns the contained widgets with the right side of this parent base. For this setting to take effect, the Layout setting must be Column.
- Default: Uses the default layout.

In the generated `*.pro` file, these settings are specified with the BASE_ALIGN_CENTER, BASE_ALIGN_TOP, BASE_ALIGN_BOTTOM, BASE_ALIGN_LEFT, and BASE_ALIGN_RIGHT keywords to the widget creation routine.

For information on other properties that control the layout of contained widgets, see # of Rows/Columns, Layout, Space, X Pad, and Y Pad.

Allow Closing

The Allow Closing attribute determines if the top-level base can be closed by the user. By default, this value is set to True and the base can be closed. To make it so the top-level base cannot be close, set this value to False.

In the generated `*.pro` file, this value is specified with the TLB_FRAME_ATTR keyword to the widget creation routine.

For information on other properties that control aspects of top-level bases, see the Allow Moving, Minimize/Maximize, System Menu, and Title Bar properties.

Note

This attribute setting is used with top-level bases only. Note that this setting is only a hint to the window system and might be ignored by some window managers.

Allow Moving

The Allow Moving attribute determines if the base can be moved. By default, this value is set to True, and the base can be moved. To suppress this behavior, set this value to False.

In the generated `*.pro` file, this value is specified with the TLB_FRAME_ATTR keyword to the widget creation routine.

For information on other attribute settings that control aspects of top-level bases, see the Allow Closing, Minimize/Maximize, System Menu, and Title Bar attributes.

Note

This attribute setting is used with top-level bases only. Note that this setting is only a hint to the window system and might be ignored by some window managers.

Floating

The Floating attribute determines if the top-level base is a floating base (always on top). By default, this setting is False, indicating that the base is *not* a floating base. To create a floating base, set this attribute to True.

If you make a top-level base floating, you must set the GROUP_LEADER keyword to a valid widget ID when calling the generated procedure.

In the generated `*.pro` file, this value is specified with the FLOATING keyword to the widget creation routine.

Note

This attribute setting is used with top-level bases only.

Grid Layout

The Grid Layout attribute determines if the base will have a grid layout, in which all columns have the same width, or in which all rows have the same height. These are the possible values:

- False: Columns or rows will not be the same size. This is the default value.
- True: Column widths or row heights are taken from the largest child widget. If you set this attribute to True, you must also set the Layout attribute to Column or Row and the # of Rows/Columns attribute to more than 1.

In the generated `*.pro` file, this value is specified with the GRID_LAYOUT keyword to the widget creation routine.

Layout

The Layout attribute specifies how components are laid out in the base. These are the possible values:

- Bulletin: Indicates that you can position the widgets anywhere on the base. This is the default setting.
- Column: Indicated that widgets should be in columns. If you set this value, you should also set the # of Rows/Columns attribute and the Alignment attribute.
- Row: Indicated that widgets should be in rows. If you set this value, you should also set the # of Rows/Columns attribute and the Alignment attribute.

Note

When using code generated by the IDL GUIBuilder on other non-Windows platforms, more consistent results are obtained by using a row or column layout for your bases instead of a bulletin board layout. By using a row or column layout, differences in the default spacing and decorations (e.g., beveling) of widgets on each platform can be avoided

The number of child widgets placed in each column or row is calculated by dividing the number of created child widgets by the number of columns or rows specified (# of Rows/Columns). When one column or row is filled, a new one is started.

The width of each column or the height of the row is determined by the largest widget in that column or row. If you set the Grid Layout attribute to True, all columns or rows are the same size; they are the size of the largest widget.

If you set the Alignment attribute for the base, the contained widgets are their “natural” size. If you do not set the Alignment attribute for the base or the child

widgets, all contained widgets will be sized to the width of the column or the height of the row.

For information on other properties that control the layout of contained widgets, see # of Rows/Columns, Alignment, Space, X Pad, and Y Pad.

In the generated `*.pro` file, this value is specified with the COLUMN or the ROW keyword to the widget creation routine.

Note

When you create a radio button or checkbox, it is created in a base, and you can add more radio buttons or checkboxes to that base (the added widgets must all be of the same type). The base in which radio buttons and checkboxes are created has a column layout setting, and buttons you add will be lined up in a column format.

Minimize/Maximize

The Minimize/Maximize attribute determines if the top-level base can be resized, minimized, and maximized. By default, this value is set to True. To disable this behavior, set this attribute to False.

In the generated `*.pro` file, this value is specified with the TLB_FRAME_ATTR keyword to the widget creation routine.

For information on other attribute settings that control aspects of top-level bases, see the Allow Closing, Allow Moving, System Menu, and Title Bar attributes.

Note

This attribute setting is used with top-level bases only. Note that this setting is only a hint to the window system and might be ignored by some window managers.

Modal

The Modal attribute determines if this top-level base is a modal dialog. By default, this value is set to False. To make the base a modal dialog, set this attribute to True.

If you set the Modal attribute to True, you cannot set the Scroll attribute, and you cannot define a menu for the top-level base. In addition, the Sensitive common attribute and the Visible base widget attribute are also disabled.

If you make a top-level base a modal dialog, you must set the GROUP_LEADER keyword to a valid widget ID in the generated procedure.

In the generated `*.pro` file, this value is specified with the MODAL keyword to the widget creation routine.

Note

This attribute setting is used with top-level bases only.

Scroll

The Scroll attribute determines if the base widget will support scrolling. By default, this attribute is set to False, and the base will not support scrolling. To give the widget scroll bars and allow for viewing portions of the widget contents that are not currently in the viewport area, set the Scroll attribute to True. In the IDL GUIBuilder, scroll bars on bases are live so that you can work on the entire virtual area of your application.

If you set the Modal attribute to True, you cannot set the Scroll attribute.

In the generated `*.pro` file, this value is specified with the SCROLL keyword to the widget creation routine.

To set the size of the scrollable region, use the X Scroll and Y Scroll attributes.

Space

The Space attribute specifies the number of pixels between the contained widgets (the children) in a column or row Layout. By default, this value is set to 3 pixels and that is the space between the contained widgets. Valid values for this attribute are 0 to *n* pixels.

In the generated `*.pro` file, this value is specified with the SPACE keyword to the widget creation routine.

To set the space from the edge of the base, use the X Pad and Y Pad properties. For information on other properties that control the layout of contained widgets, see # of Rows/Columns, Alignment, and Layout.

Note

You cannot set this attribute on a base containing radio buttons or checkboxes.

System Menu

The System Menu attribute determines if the system menu is displayed or suppressed on a top-level base. By default, this value is set to True, indicating that the system menu will be used. To suppress the menu, set this attribute to False.

In the generated `*.pro` file, this value is specified with the TLB_FRAME_ATTR keyword to the widget creation routine.

For information on other attribute settings that control aspects of top-level bases, see the Allow Closing, Allow Moving, Minimize/Maximize, and Title Bar attributes.

Note

This attribute setting is used with top-level bases only.

Title

The Title attribute specifies the title of a top-level base. By default, this value is set to IDL, but you can change it to any string.

In the generated `*.pro` file, this value is specified with the TITLE keyword to the widget creation routine.

Note

This attribute setting is used with top-level bases only.

Title Bar

The Title Bar attribute determines if the title bar will be displayed. By default, this value is set to True, and the title bar is displayed. To suppress the display of the title bar, set this value to False.

In the generated `*.pro` file, this value is specified with the TLB_FRAME_ATTR keyword to the widget creation routine.

For information on other attribute settings that control aspects of top-level bases, see the Allow Closing, Allow Moving, Minimize/Maximize, and System Menu attributes.

Note

This attribute setting is used with top-level bases only. Note that this setting is only a hint to the window system and might be ignored by some window managers.

Visible

The Visible attribute specifies whether to show or hide the base component and its descendants. Show, the default value, specifies to display the hierarchy when realized. The Hide value specifies that the hierarchy should *not* be displayed initially. This mapping operation applies only to base widgets.

In the generated `*.pro` file, this value is specified with the MAP keyword to the widget creation routine.

Note

If you set the Modal attribute to True, you cannot set the Visible attribute value.

X Pad

The X Pad attribute specifies the horizontal space (in pixels) between child widgets and the edges of rows or columns. By default, this value is set to 3 pixels, indicating that there are 3 pixels between the edge of the base and the contained widgets. Valid values for this attribute are 0 to *n* pixels.

In the generated `*.pro` file, this value is specified with the XPAD keyword to the widget creation routine.

To set the space between widgets, use the Space attribute. For information on other attributes that control the layout of contained widgets, see # of Rows/Columns, Alignment, Layout, and Y Pad.

Note

You cannot set this attribute for a base that contains radio buttons or checkboxes. In the IDL GUIBuilder, a base is created when you add a radio button or checkbox to an interface, and you can add more radio buttons or checkboxes to that base. When you add the buttons, they are lined up in a column format.

X Scroll

The X Scroll attribute specifies the width in pixels of the base area, which includes the exposed as well as the virtual area. There is no default value set, but you can set this value to any number of pixels from 0 to *n*. To add scroll bars to the base, use the Scroll attribute, and to set the height of the scrollable base area, use the Y Scroll attribute.

In the generated `*.pro` file, this value is specified with the XSIZE keyword to the widget creation routine.

Note

To set the width of the displayed widget, use the X Size common attribute.

Y Pad

The Y Pad attribute specifies the vertical space (in pixels) between child components and the edge of the base in a row or column Layout. By default, this value is set to 3 pixels, indicating that there are 3 pixels between the edge of the base and the contained widgets. Valid values for this attribute are 0 to *n* pixels.

In the generated `*.pro` file, this value is specified with the YPAD keyword to the widget creation routine.

To set the space between widgets, use the Space attribute. For information on other attributes that control the layout of contained widgets, see # of Rows/Columns, Alignment, Layout, and X Pad.

Note

You cannot set this attribute on a base containing radio buttons or checkboxes. In the IDL GUIBuilder, a base is created when you add a radio button or checkbox to an interface, and you can add more radio buttons or checkboxes to that base.

Y Scroll

The Y Scroll attribute specifies the height in pixels of the base area, which includes the exposed as well as the virtual area. There is no default value set, but you can set this value to any number of pixels from 0 to *n*.

To add scroll bars to the base, use the Scroll attribute, and to set the width of the base area, use the X Scroll attribute.

In the generated `*.pro` file, this value is specified with the YSIZE keyword to the widget creation routine.

Note

To set the height of the displayed widget, use the Y Size common attribute.

Base Widget Events

For base widgets, you can set common event properties and base-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see “Common Events” on page 651.

The following is a list of event properties specific to base widgets:

OnContextEvent

The OnContextEvent value is the routine name that is called when the user clicks the right-hand mouse button over the base widget. In the generated `*_eventcb.pro` file, the event calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the returned event structure returned when the user clicks the right-hand mouse button and is of the following type:

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

The X and Y fields give the device coordinates at which the event occurred, measured from the upper left corner of the base widget.

OnFocus

The OnFocus value is the routine name that is called when the keyboard focus of the base changes. In the generated `*_eventcb.pro` file, the event calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when the keyboard focus changes and is of the following type:

```
{ WIDGET_KBRD_FOCUS, ID:0L, TOP:0L, HANDLER:0L, ENTER:0 }
```

ENTER returns 1 if the base is gaining the keyboard focus, and returns 0 if the base is losing the keyboard focus.

OnIconify

The OnIconify value is the routine that is called when the user iconifies or restores the top-level base widget. In the generated `*_eventcb.pro` file, the event calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when a user iconifies or restores the widget using the window manager and is of the following type:

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

ICONIFIED is 1 (one) if the user iconified the base and 0 (zero) if the user restored the base.

Note

This event procedure is valid for top-level bases only.

OnKillRequest

The OnKillRequest value is the routine that is called when the user attempts to kill the top-level base widget. In the generated `*_eventcb.pro` file, the event calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when a user tries to destroy the widget using the window manager and is of the following type:

```
{ WIDGET_KILL_REQUEST, ID:0L, TOP:0L, HANDLER:0L }
```

Note that this event structure contains the standard three fields that all widgets contain.

Note

This event procedure is valid for top-level bases only.

OnMove

The OnMove value is the routine that is called when the user moves the top-level base widget. In the generated `*_eventcb.pro` file, the event calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when a user moves the widget and is of the following type:

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

X and Y are the new location of the top left corner of the base.

Note

This event procedure is valid for top-level bases only.

OnSizeChange

The OnSizeChange value is the name of the routine that is called when the top-level base has been resize. In the generated `*_eventcb.pro` file, the event calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when the top-level base is resized by the user and is of the following type:

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

The X and Y fields return the new width of the base, not including any frame provided by the window manager.

Note

This event procedure is valid for top-level bases only.

Button Widget Properties

In IDL, a button widget can be a button (push button), radio button, or checkbox.

A push button is activated by a single-click. Push buttons can be of any size. You can set the Menu attribute to yes for a button widget, and then it can contain a pull-down menu. When you do so, the Label is enclosed in a box to indicate that the button is a menu button.

Radio buttons have two states, set and unset, and they belong to a group that allows only one radio button selection for that group. The group is defined as all buttons contained in the same exclusive base widget. When a radio button in a base (in a group) is selected, any other button selection in that base is cleared. When you create a radio button in the IDL GUIBuilder, it is created in an exclusive base widget, and you can add only radio buttons to that base.

Checkboxes have two states, set and unset, and they are grouped in a non-exclusive base widget. The base widget allows for any number of checkboxes to be set at one time, and you can also use single checkboxes in your interface. When you create a checkbox in the IDL GUIBuilder, it is created in an non-exclusive widget base, and you can add only checkboxes to this base.

When programming in IDL, you create push buttons, radio buttons, and checkboxes using the WIDGET_BUTTON function. For more information, see "WIDGET_BUTTON" in the *IDL Reference Guide* manual.

Note

The bases in which radio buttons and checkboxes are created have the Layout attribute set to column so when you add more widgets they are lined up appropriately.

Creating Multiple Radio Buttons or Checkboxes

To create several radio buttons or checkboxes in a base widget:

1. Click on the radio button or checkbox tool, and click on the location to add the button. This creates a base with one radio button or checkbox in it.
2. Click on the radio button or checkbox tool, and click in the radio button or checkbox base area you just created. This adds a radio button or checkbox to the base.

When you drop a button in an exclusive or non-exclusive base, the added buttons line up in columns; by default, these exclusive and non-exclusive bases have their Layout attribute set to Column.

3. Repeat step 2 until you have the desired number of buttons.
4. If you want to change the layout of the checkboxes or radio buttons, you can open the Properties dialog and set the Layout common attribute for the base widget to Row or Bulletin.
5. To set the properties for each button in the base, open the Properties dialog, click the push pin button to keep it on top, then click on each radio button or checkbox to set their individual properties.

Button, Radio Button, and Checkbox Widget Attributes

For button widgets, you can set common attributes and button-specific attributes. For a list of common attributes, see “Common Attributes” on page 648. The following is a list of button widget attributes, which apply to push buttons, radio buttons, and/or checkboxes:

Alignment

The Alignment attribute specifies how the text label is aligned in the button widget. These are the possible alignment values:

- Center: The label text is centered.
- Left: The label text is left-justified.
- Right: The label text is right-justified.

In the generated `*.pro` file, this value is specified by the ALIGN_CENTER, the ALIGN_LEFT, or the ALIGN_RIGHT keyword to the widget creation routine.

Bitmap

The Bitmap attribute allows you to select a bitmap to be displayed in the push button, and it allows you to access the Bitmap Editor to create or modify a bitmap file (*.bmp file). This value applies only to buttons (not to radio buttons or checkboxes).

To set this value, set the Type value to Bitmap, then the Bitmap attribute displays in the Properties dialog. When the button type is “Bitmap”, you can set the Bitmap attribute to the path and name of the bmp file.

When you click on the arrow in the Bitmap attribute Value field, you can choose from the following options:

- Select Bitmap: Launches an Open dialog that you can use to locate and select the existing *.bmp file to be placed in the button.
- Edit Bitmap: Launches an Open dialog that you can use to locate and select the existing *.bmp file to be opened in the Bitmap Editor. You can modify the bitmap and save it. The bitmap is then displayed in the button.
- New Bitmap: Opens the Bitmap Editor which you can use to create and save a bitmap. When you save the new bitmap, it is displayed in the button.

In the generated `*.pro` file, this value is specified with the VALUE and Bitmap keyword to the widget creation routine.

For information on using the Bitmap Editor, see "Using the Bitmap Editor" on page 624.

Label

The Label attribute specifies the text label for a button. If you set the Type attribute to Bitmap (for push buttons only), this value is not displayed. For radio buttons and checkboxes, the label value is the text string displayed next to the button. By default, this value is set to Button, and you can change it to any string.

In the generated `*.pro` file, this value is specified with the VALUE keyword to the widget creation routine.

No Release

The No Release attribute enables and disables the dispatching of button release events for radio buttons and checkboxes. Normal buttons do not generate events when released, but radio buttons and checkboxes can return separate events for the select and release actions. These are the possible values:

- True: The release event is not returned; only the select event is returned. This is the default setting.
- False: Both the release and select events are returned.

In the generated `*.pro` file, this values is specified with the NO_RELEASE keyword to the widget creation routine.

Note

The No Release attribute is for radio buttons and checkboxes only.

Tooltip

The Tooltip attribute specifies a short text string that will be displayed when the mouse cursor is positioned over the button. The length of the tooltip is not explicitly limited, but since tooltips are displayed in a single line, it is best to keep the text short.

Type

The Type attribute specifies the type of push button.

Note

This attribute applies only to push buttons, not to radio buttons or checkboxes.

The possible values are:

- Push: The button widget is a plain push button. This is the default value.
- Menu: The button contains a menu. After you select this value, you can right-click on the button widget, choose Edit Menu, and define a menu to display, using the Menu Editor. See "Using the Menu Editor" on page 621 for details on using the Menu Editor.
- Bitmap: The button displays a bitmap, which you would use to create a toolbar for example. If you change the Type value to Bitmap, the Bitmap attribute is displayed and you can select, modify, or create a bitmap to display on the button.

In the generated `*.pro` file, this value is specified with the MENU or VALUE keywords to the widget creation routine.

Button, Radio Button, and Checkbox Widget Events

For button widgets, you can set common event properties and button-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see "Common Events" on page 651.

The following is the event property specific to button widgets; it applies to push buttons, radio buttons, and checkboxes:

OnButtonPress

The OnButtonPress value is the routine that is called when the button is pressed, or when a button is released for a radio button or checkbox button. In the generated `*_eventcb.pro` file, the event calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the returned event structure, which is of the following type:

```
{ WIDGET_BUTTON, ID:0L, TOP:0L, HANDLER:0L, SELECT:0 }
```

SELECT is set to 1 if the button was set, and 0 if released. Push buttons do not generate events when released, so SELECT will always be 1 for a push button. However, radio buttons and checkboxes are toggle buttons, and thus return separate events for the set and the release actions. To control whether or not release events are returned, set the No Release attribute.

Text Widget Properties

Use text widgets to display text, and optionally, use them to accept textual input from users. The text widgets can have one or more lines, and if necessary, the widget can contain scroll bars to allow for viewing longer text.

When programming in IDL, you create text widgets using the WIDGET_TEXT function. For more information, see "WIDGET_TEXT" in the *IDL Reference Guide* manual.

Note

Use text widgets for displaying large amounts of text, or when you want the user to be able to edit the text. Use label widgets to display single-line labels that the user cannot edit.

Text Widget Attributes

For text widgets, you can set common attributes and text-specific attributes. For a list of common attributes, see "Common Attributes" on page 648. The following are the attributes specific to text widgets:

Editable

The Editable attribute determines if the text widget is editable or not. By default, this value is set to False, which means the text widget is not editable. To make the text widget editable, set this value to True.

In the generated `*.pro` file, this value is specified with the EDITABLE keyword to the widget creation routine.

Height

The Height attribute specifies the height of the text widget in text lines. Valid values for this attribute are 1 to *n*. The default value, is 1, or one text line.

Note that the physical height of the text widget depends on the value of the Height attribute and on the size of the font used. The default font size is used, unless you modify your generated code to use a different font, and the default font size is platform specific.

In the generated `*.pro` file, this value is specified by the YSIZE keyword to the widget creation routine.

Initial Value

The Initial Value attribute specifies the initial array of values that are placed in the text widget. You can enter either a string or an array of strings.

To enter more than one string in the Value field, type in a string, then press Control+Enter (at the end of each line). This moves you to the next line. When you have entered the strings you want, press Enter to set the values.

In the generated `*.pro` file, this value is specified by the VALUE keyword to the widget creation routine.

Note

Variables returned by the GET_VALUE keyword to WIDGET_CONTROL are always string arrays, even if a scalar string is specified in the call to WIDGET_TEXT.

Scroll

The Scroll attribute determines if the text widget displays scroll bars. By default, this value is set to False, which indicates that no scroll bars will be displayed. To have the text widget display scroll bars, set this value to True.

In the generated `*.pro` file, this value is specified by the SCROLL keyword to the widget creation routine.

Width

The Width attribute specifies the width of the text widget in characters. Valid values for this attribute are 0 to *n*. By default, Width is set to 0, which indicates that default IDL sizing should be used when, as long as default Component Sizing is also set.

Note that the physical width of the text widget depends on the value of the Width attribute and on the size of the font used. The default font size varies according to your windowing system. On Windows, the default size is roughly 20 characters. On Motif, the default size depends on the system default.

In the generated `*.pro` code, this value is specified with the XSIZE keyword.

Word Wrapping

The Word Wrapping attribute determines whether scrolling or multi-line text widgets should automatically break lines between words to keep the text from extending past the right edge of the text display area. By default this value is set to False, and carriage returns are not automatically entered; the value of the text widget will remain

a single-element array. To have the text widget enter carriage returns at the end of lines, change this value to True.

In the generated `*.pro` code, this value is specified with the WRAP keyword.

Text Widget Events

For text widgets, you can set common event properties and text-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see "Common Events" on page 651.

You can set the following event values for text widgets:

OnContextEvent

The OnContextEvent value is the routine name that is called when the user clicks the right-hand mouse button over the text widget. In the generated `*_eventcb.pro` file, the event calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when the user clicks the right-hand mouse button and is of the following type:

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

The X and Y fields give the device coordinates at which the event occurred, measured from the upper left corner of the text widget.

OnDelete

The OnDelete value is the routine that is called when text is deleted from the text widget. To set this event value, you must set the Editable attribute to True.

In the generated `*_eventcb.pro` file, the calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when any amount of text is deleted from a text widget. The event structure is of the following type:

```
{ WIDGET_TEXT_DEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:2, OFFSET:0L,
   LENGTH:0L }
```

OFFSET is the (zero-based) character position of the first character to be deleted, and it is also the insertion position that will result when the characters have been deleted.

LENGTH gives the number of characters deleted, where 0 (zero) indicates that no characters were deleted.

OnFocus

The OnFocus value is the routine that is called when the keyboard focus changes. In the generated `*_eventcb.pro` event code, the calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the returned structure, which is of the following type:

```
{ WIDGET_KBRD_FOCUS, ID:0L, TOP:0L, HANDLER:0L, ENTER:0 }
```

ENTER returns 1 if the text widget is gaining the keyboard focus, or 0 if the text widget is losing the keyboard focus.

OnInsertCh

The OnInsertCh value is the routine that is called when a single character is inserted in the widget. To set this event value, you must set the Editable attribute to True.

In the generated `*_eventcb.pro` file, the calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when a single character is typed or pasted into a text widget by a user. The event structure is of the following type:

```
{ WIDGET_TEXT_CH, ID:0L, TOP:0L, HANDLER:0L, TYPE:0, OFFSET:0L,
   CH:0B }
```

OFFSET is the (zero-based) insertion position that will result after the character is inserted. CH is the ASCII value of the character.

OnInsertString

The OnInsertString value is the routine that is called when a text string is inserted in the text widget. To set this event value, you must set the Editable attribute to True.

In the generated `*_eventcb.pro` file, the calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when multiple characters are inserted in to text widget. The event structure is of the following type:

```
{ WIDGET_TEXT_STR, ID:0L, TOP:0L, HANDLER:0L, TYPE:1, OFFSET:0L,
```

```
STR:'' }
```

OFFSET is the (zero-based) insertion position that will result after the text is inserted. STR is the string to be inserted.

OnTextSelect

The OnTextSelect value is the routine that is called when text is selected in the text widget. To set this event value, you must also set the Editable attribute to True.

In the generated `*_eventcb.pro` file, the calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when an area of text is selected. The event structure is of the following type:

```
{ WIDGET_TEXT_SEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:3, OFFSET:0L,
   LENGTH:0L }
```

This event announces a change in the insertion point. OFFSET is the (zero-based) character position of the first character selected, which can also be the insertion position. LENGTH gives the number of characters involved, where zero indicates that no characters are selected.

Note

Text insertion, text deletion, or any change in the current insertion point causes any current selection to be lost. In such cases, the loss of selection is implied by the text event reporting the insert, delete, or movement event, and a separate zero length selection event is *not* sent.

Label Widget Properties

Label widgets display static text. They are similar to single-line text widgets, but they are optimized for small labeling purposes.

There are no label widget-specific event properties.

When programming in IDL, you create label using the WIDGET_LABEL function. For more information, see "WIDGET_LABEL" in the *IDL Reference Guide* manual.

Note

Use label widgets to display single-line labels that you do not want the user to be able to edit. Use text widgets for displaying larger amounts of text, or text that you want the user to be able to edit.

Label Widget Attributes

For label widgets, you can set common attributes and label-specific attributes. For a list of common attributes, see "Common Attributes" on page 648. These are the label widget attributes:

Alignment

The Alignment attribute specifies how label Text is aligned. These are the possible values:

- Left: The text is left-justified. This is the default value.
- Center: The text is centered.
- Right: The text is right-justified.

In the generated `*.pro` file, this value is specified with the ALIGN_CENTER, the ALIGN_RIGHT, or the ALIGN_LEFT keyword to the widget creation routine.

Sunken

The Sunken attribute specifies whether the label should be displayed with a "sunken" border. Sunken borders are often used for status lines and other interface elements in which the text of the label changes based on user actions or the state of the application.

Text

The Text attribute specifies the text string that is displayed in the label widget. By default, this value is set to Label, and you can set it to any string.

In the generated `*.pro` file, this value is specified with the VALUE keyword to the widget creation routine.

Label Widget Events

There are *no* events specific to Label widgets. For a list of the common widget events, see “Common Events” on page 651.

Slider Widget Properties

Horizontal or vertical slider widgets allow for the selection of a value within a range of possible integer values. A slider widget is a rectangular region representing a range of values, with a sliding pointer inside that indicates or selects the current value. This sliding pointer can be manipulated by the user dragging it with the mouse, or within IDL code.

When programming in IDL, you create horizontal or vertical slider widgets using the WIDGET_SLIDER function. See "WIDGET_SLIDER" in the *IDL Reference Guide* manual.

Horizontal and Vertical Slider Widget Attributes

For slider widgets, you can set common attributes and slider-specific attributes. For a list of common attributes, see "Common Attributes" on page 648. The following is a list of slider attributes:

Maximum Value

The Maximum Value attribute specifies the maximum range value for the slider. The default value is 100, but you can set this attribute to any integer. This value works with the Minimum Value attribute.

In the generated `*.pro` file, this value is specified with the MAXIMUM keyword to the widget creation routine.

Minimum Value

The Minimum Value attribute specifies the minimum range value of the slider. The default value is 0, but you can set this attribute to any integer. This attribute works with the Maximum Value attribute.

In the generated `*.pro` file, this value is specified with the MINIMUM keyword to the widget creation routine.

Position

The Position attribute specifies the initial value position of the slider. By default this is set to 0, so the initial position will be at 0. You can set this value to any integer within the range of the Maximum Value and Minimum Value attribute settings.

In the generated `*.pro` file, this value is specified with the VALUE keyword to the widget creation routine.

Suppress Value

The Suppress Value attribute controls the display of the current slider value. Sliders work only with integer units. You can use this attribute to suppress the actual value of a slider so that a program can present the user with a slider that seems to work in other units (such as floating-point) or with a non-linear scale. By default, this value is set to False, indicating that the current value, in integer units, should be displayed. To suppress the display of the current values, set this attribute value to True.

In the generated `*.pro` file, this value is specified with the SUPPRESS_VALUE keyword to the widget creation routine.

Title

The Title attribute specifies the label or title that is associate with the slider widget. By default, this is not set; it is an empty string. You can set the title to any string.

In the generated `*.pro` file, this value is specified with the TITLE keyword to the widget creation routine.

Horizontal and Vertical Slider Widget Events

For slider widgets, you can set common event properties and slider-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see “Common Events” on page 651.

This is the event property specific to slider widgets:

OnChangeValue

The OnChangeValue specifies the routine that is called when the value of the slider is changed. When you set this event value, the calling sequence looks like this in the generated `*_eventcb.pro` file:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when a slider is moved. The event structure is of the following type:

```
{ WIDGET_SLIDER, ID:0L, TOP:0L, HANDLER:0L, VALUE:0L, DRAG:0 }
```

VALUE returns the new value of the slider. DRAG returns integer 1 if the slider event was generated as part of a drag operation, or zero if the event was generated when the user had finished positioning the slider. Note that the slider widget only generates events during the drag operation if the DRAG keyword is set, and if the application is running on Motif. That is, in most cases, DRAG will return zero.

Droplist Widget Properties

Droplist widgets display a single entry from a list of possible choices. To choose from the list, click the droplist, then click on the item in the list. On Motif operating systems, the droplist widget looks like a button, which when clicked displays the drop-down list.

When programming in IDL, you create droplist widgets using the WIDGET_DROPLIST function. For more information, see "WIDGET_DROPLIST" in the *IDL Reference Guide* manual.

Droplist Widget Attributes

For droplist widgets, you can set common attributes and droplist-specific attributes. For a list of common attributes, see "Common Attributes" on page 648. These are the droplist attributes:

Initial Value

The Initial Value attribute specifies the initial list of values that are placed in the droplist widget. The initial value of a droplist can be a scalar string, or it can be a list of strings. By default, this value is not set, and the droplist is empty.

To enter more than one string in the Value field, type in a string, then press Control+Enter (at the end of each line). This moves you to the next line. When you have entered as many strings as you want, press Enter to set the values.

In the generated `*.pro` file, this value is specified with the VALUE keyword to the widget creation routine.

Title

The Title attribute specifies the title string, or label, for the droplist. This value can be any string. By default, this value is set to NULL.

In the generated `*.pro` file, this value is specified by the TITLE keyword to the widget creation routine.

Droplist Widget Events

For droplist widgets, you can set common event properties and droplist-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see "Common Events" on page 651.

This is the event property specific to droplist widgets:

OnSelectValue

The OnSelectValue specifies the routine that is called when a droplist item is selected. When a user selects an item from a droplist, the widget deselects the previously selected item, changes the visible item on the droplist, and generates an event.

When you set this event value, the calling sequence looks like this in the generated `*_eventcb.pro` file:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when a user selects an item from a droplist. The event structure is of the following type:

```
{ WIDGET_DROPLIST, ID:0L, TOP:0L, HANDLER:0L, INDEX:0L }
```

INDEX returns the index of the selected item. This value can be used to index the array of names originally used to set the widget's value.

Note

On some platforms, when a droplist widget contains only one item and the user selects it again, the action does note not generate an event. Events are always generated on selection actions if the list contains multiple items.

Listbox Widget Properties

The listbox displays a list of text items from which a user can select, by clicking on them. The listboxes have vertical scroll bars to allow viewing of a long list of items.

When programming in IDL, you create listbox widgets using the WIDGET_LIST function. For more information, see "WIDGET_LIST" in the *IDL Reference Guide* manual.

Listbox Widget Attributes

For listbox widgets, you can set common attributes and listbox-specific attributes. For a list of common attributes, see "Common Attributes" on page 648. These are the listbox widget attributes:

Height

The Height attribute specifies the height of the listbox based on the number of lines that are visible. The possible values for the attribute are 1 to *n*. By default, Height is set to 1, which indicates the default size of one line will be used.

Note that the final size of the widget may be adjusted to include space for scroll bars, which are not always visible, so the listbox might be slightly larger than specified.

In the generated `*.pro` file, this value specified with the YSIZE keyword to the widget creation routine.

Initial Value

The Initial Value attribute specifies the initial list of values that are placed in the list widget. By default, the list is empty, but you can set this value to a scalar string or a list of strings. List widgets are sized based on the length (in characters) of the longest item specified in the array of values.

To enter more than one string in the Value field, type in a string, then press Control+Enter (at the end of each line). This moves you to the next line. When you have entered as many strings as you want, press Enter to set the values.

In the generated `*.pro` file, this value is specified by the VALUE keyword to the widget creation routine.

Multiple

The Multiple attribute determines if the user can select multiple list items. By default, the setting is False, which allows for only one selection. To enable multiple list item

selection, set this value to True. Multiple selections are handled using the method appropriate to the platform the application is running on.

In the generated `*.pro` file, this value is specified with the MULTIPLE keyword to the widget creation routine.

Width

The Width attribute specifies the width of the listbox in characters. The possible values for the attribute are 0 to *n*. By default, Width is set to 0, which indicates that default sizing will be used, as long as the Component Sizing attribute is set to default.

By default, IDL sizes widgets to fit the situation. However, if the desired effect is not produced, use explicit Component Sizing with the Width attribute to set your own sizing. The final size of the widget may be adjusted to include space for the scroll bar, which is not always visible, so your widget may be slightly larger than specified.

In the generated `*.pro` file, this value specified with the XSIZE keyword to the widget creation routine.

Listbox Widget Events

For listbox widgets, you can set common event properties and listbox-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see “Common Events” on page 651.

The following is the event property specific to listbox widgets:

OnContextEvent

The OnContextEvent value is the routine name that is called when the user clicks the right-hand mouse button over the list widget. In the generated `*_eventcb.pro` file, the event calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when the user clicks the right-hand mouse button and is of the following type:

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

The X and Y fields give the device coordinates at which the event occurred, measured from the upper left corner of the base widget.

OnSelectValue

The OnSelectValue specifies a valid IDL routine name that is called when a list item is selected. When a user clicks on an item in the listbox to select the item, an event is generated.

When you set this event value, the calling sequence looks like this in the generated `*_eventcb.pro` file:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the returned event structure, which is of the following type:

```
{ WIDGET_LIST, ID:0L, TOP:0L, HANDLER:0L, INDEX:0L, CLICKS:0L }
```

The first three fields are the standard fields found in every widget event. INDEX returns the index of the selected item. This index can be used to subscript the array of names originally used to set the widget's value. CLICKS returns either 1 or 2, depending on how the list item was selected. If the list item is double-clicked, CLICKS is set to 2.

Note

If you are writing a widget application that requires the user to double-click on a list widget, you will need to handle two events. The CLICKS field will return a 1 on the first click and a 2 on the second click.

Draw Widget Properties

Draw widgets are rectangular regions that IDL treats as standard graphics windows. Use draw widgets to display either IDL Direct graphics or IDL Object graphics, depending on the value of the Graphics Type attribute. You can direct any graphical output that can be produced by IDL to one of these widgets, either by using the WSET function or by using the object reference of a draw widget's IDLgrWindow object.

Draw widgets can contain scroll bars that allow for viewing of a graphical region larger than the area containing the widget.

When programming in IDL, you create draw area widgets using the WIDGET_DRAW function. For more information, see "WIDGET_CONTROL" in the *IDL Reference Guide* manual.

Draw Area Widget Attributes

For a draw area widget, you can set common attributes and draw area-specific attributes. For a list of common attributes, see "Common Attributes" on page 648. These are the draw area-specific attributes:

Color Model

The Color Model attribute specifies the color model that should be used for displaying information on the draw widget. This attribute value is used only when the Graphics Type attribute is set to Object, for IDL Object Graphics. These are the possible values for the Color Model attribute:

- Index: The draw widget's associated IDLgrWindow object uses indexed color.
- RGB: The RGB color model is used. This is the default value.

In the generated `*.pro` file, this value is specified by the COLOR_MODEL keyword to the widget creation routine.

Colors

The Colors attribute specifies the number of colors that the drawable should attempt to use from the system color table. This attribute is only valid with the Graphics Type attribute is set to Direct, for IDL Direct Graphics. By default, the Color attribute is set to 0, which indicates that IDL will attempt to get all available colors. That is, all or most of the available color indices are allocated, based on the window system in use.

You can set the Colors attribute to any integer, but most values will be in the range of -256 < *n* < 256.

This attribute has effect only if it is supplied when the first IDL graphics window is created. To use monochrome windows on a color display, set the Colors attribute to 2 for the first window. One color table is maintained for all running IDL windows.

In the generated `*.pro` file, this value is specified by the COLORS keyword to the widget creation routine.

Graphics Type

The Graphics Type attribute specifies the type of graphics that the draw widget will support. These are the possible values:

- Direct: The draw widget will display Direct Graphics. This is the default value. The Colors attribute is used only when Graphics Type is set to Direct.
- Object: The draw widget will display IDL Object Graphics. The Color Model and Renderer properties are used only when the Graphics Type is set to Object.

In the generated `*.pro` file, this value is specified with the GRAPHICS_LEVEL keyword to the widget creation routine.

Renderer

The Renderer attribute specifies which graphics renderer to use with IDL Object Graphics. That is, for this attribute to be used, the Graphics Type attribute should be set to Object. These are the possible values for the Renderer attribute:

- OpenGL: The platform's native OpenGL renderer is used when drawing objects within the window. If your platform does not have a native OpenGL implementation, IDL's software implementation is used as the renderer. This value is set by default.
- Software: IDL's software implementation is used when drawing objects within the window.

In the generated `*.pro` file, this value is specified by the RENDERER keyword to the widget creation routine.

For more information, see "Hardware vs. Software Rendering" in Chapter 34 of the *Using IDL* manual.

Note

The renderer selection can also affect the maximum size of a draw widget.

Retain

The Retain attribute specifies how backing store is performed in the draw area. These are the possible values:

- None: There is no backing store. When the Retain attribute is set to None, you should track OnExpose events so that you can handle the redrawing of the screen. This is the default value.
- System: The server or window system should provide backing store.
- IDL Pixmap: IDL should provide backing store.

In the generated `*.pro` file, this value is specified with the RETAIN keyword to the widget creation routine.

For information on the use of the Retain attribute with Direct Graphics, see "Backing Store" in Appendix A, "IDL Graphics Devices" in the *IDL Reference Guide* manual. For more information on this attribute with IDL Object Graphics, see "IDLgrWindow::Init" in the *IDL Reference Guide* manual.

Scroll

The Scroll attribute specifies if the draw area widget will support scrolling, and will have scroll bars. By default, this value is set to False, which indicates there are no scroll bars. To display scroll bars, and enable scrolling, set this value to True. If you do so, set the size of the scrollable area with the X Scroll and Y Scroll properties.

In the generated `*.pro` file, this value is specified with the SCROLL keyword to the widget creation routine.

Tooltip

The Tooltip attribute specifies a short text string that will be displayed when the mouse cursor is positioned over the draw area. The length of the tooltip is not explicitly limited, but since tooltips are displayed in a single line, it is best to keep the text short.

X Scroll

The X Scroll attribute specifies the width in pixels of the drawing area. This width includes the exposed and virtual area. By default, this value is not set. You can set X Scroll to any width from 0 to *n*. If you set this value, also set the Scroll and Y Scroll attribute values.

In the generated `*.pro` file, this value is specified with the XSIZE keyword to the widget creation routine.

Note

To set the width of the displayed widget, use the X Size common attribute.

Y Scroll

The Y Scroll attribute specifies the height in pixels of the drawing area. This height includes the exposed and virtual area. By default, this value is not set. You can set Y Scroll to any height in pixels from 0 to *n*. If you set this value, also set the Scroll and X Scroll properties.

In the generated `*.pro` file, this value is specified with the YSIZE keyword to the widget creation routine.

Note

To set the height of the displayed widget, use the Y Size common attribute.

Draw Area Widget Events

For draw area widgets, you can set common event properties and draw area-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see "Common Events" on page 651.

These are the draw area event properties:

OnButton

The OnButton value is the routine that is called when a mouse button event is detected. In the generated `*_eventcb.pro` file, the calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the returned event structure, which is of the following type:

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

Note that this is the same event structure returned for all draw area events; OnButton, OnExpose, OnMotion, and OnViewportMoved events all return the same structure. Therefore the following paragraphs describe all these events.

TYPE returns a value that describes the type of draw widget interaction that generated an event. If there is a button press, it returns 0, and if there is a button release, it returns 1. If there is motion, it returns 2 (for an OnMotion event). If the

viewport moved with the scroll bars, it returns 3 (for an OnViewportMoved event). If the visibility changes, it returns 4 (for an OnExpose event).

The X and Y fields give the device coordinates at which the event occurred, measured from the lower left corner of the drawing area.

PRESS and RELEASE are bitmasks in which the least significant bit represents the left-most mouse button. The corresponding bit of PRESS is set when a mouse button is pressed, and in RELEASE when the button is released. If the event is a motion event, both PRESS and RELEASE returns zero.

CLICKS returns either 1 or 2. If the time interval between button-press events is greater than the time interval for a double-click event for the system, the CLICKS field returns 1. If the time interval between two button-press events is less than the time interval for a double-click event for the platform, the CLICKS field returns 2.

The MODIFIERS field is valid for button press, button release, motion, and keyboard events. It is a bitmask which returns the current state of several keyboard modifier keys at the time the event was generated. If a bit is zero, the key is up. If the bit is set, the key is depressed. See "Widget Events Returned by Draw Widgets" in the *IDL Reference Guide* manual for complete details.

Keyboard events are generated with the value of the TYPE field equal to 5 or 6. If the event was generated by an ASCII keyboard character, the TYPE field will be set to 5 and the ASCII value of the key will be returned in the CH field. (Note that ASCII values can be converted to the string representing the character using the IDL STRING routine.) If the event was generated due to a non-ASCII keyboard character, the type of the event will be set to 6 and a numeric value representing the key will be returned in the KEY field. See "Widget Events Returned by Draw Widgets" in the *IDL Reference Guide* manual for complete details.

OnExpose

The OnExpose value is the routine that is called when the visibility of any portion of the draw window (or viewport) changes or is exposed. In the generated `*_eventcb.pro` file, the calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the returned event structure, which is of the following type:

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

Note that this is the same event structure returned for all draw area events; OnButton, OnExpose, OnKeyboard, OnMotion, and OnViewportMoved events all return the same structure. For information on this structure, see OnButton.

OnKeyboard

The OnKeyboard value is the routine that is called when the user presses a key on the keyboard while the draw window has focus. In the generated `*_eventcb.pro` file, the calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the returned event structure, which is of the following type:

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

Note that this is the same event structure returned for all draw area events; OnButton, OnExpose, OnKeyboard, OnMotion, and OnViewportMoved events all return the same structure. For information on this structure, see OnButton.

OnMotion

The OnMotion value is the routine that is called when a mouse motion event is detected. In the generated `*_eventcb.pro` file, the calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the returned event structure, which is of the following type:

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

Note that this is the same event structure returned for all draw area events; OnButton, OnExpose, OnKeyboard, OnMotion, and OnViewportMoved events all return the same structure. For information on this structure, see OnButton.

OnViewportMoved

The OnViewportMoved value is the routine that is called when the viewport of a scrolling draw widget is moved, using the scroll bars. In the generated `*_eventcb.pro` file, the calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the returned event structure, which is of the following type:

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

Note that this is the same event structure returned for all draw area events; OnButton, OnExpose, OnKeyboard, OnMotion, and OnViewportMoved events all return the same structure. For information on this structure, see OnButton.

Table Widget Properties

Table widgets display data and allow for data editing by the user. Tables can have one or more rows and one or more columns.

When programming in IDL, you create table widgets using the WIDGET_TABLE function. For more information, see "WIDGET_TABLE" in the *IDL Reference Guide* manual.

Table Widget Attributes

For table widgets, you can set common attributes and table-specific attributes. For a list of common attributes, see "Common Attributes" on page 648. These are the table widget-specific attributes:

Alignment

The Alignment attribute specifies how the text is aligned in the cells. These are the possible values:

- Left: The text is left-justified. This is the default value.
- Right: The text is right-justified.
- Center: The text is centered.

In the generated `*.pro` file, this value is specified with the ALIGNMENT keyword to the widget creation routine.

Column Labels

The Column Labels attribute specifies the labels for the table columns. By default, this value is set to empty strings, but you can set it to any set of strings. To set the labels for table rows, use the Row Labels attribute.

To enter more than one string in the Value field, type in a string, then press Control+Enter (at the end of each line). This moves you to the next line, or the next label for a column. When you have entered as many labels as you want, press Enter to set the values.

In the generated `*.pro` file, this value is specified with the COLUMN_LABELS keyword to the widget creation routine.

Disjoint Selection

The Disjoint Selection attribute determines whether the user is allowed to select multiple, unconnected cell regions within the table. By default, this value is set to False, indicating that only a single selection is allowed. To allow disconnected regions, set this value to True.

In the generated `*.pro` file, the False value is specified with the DISJOINT_SELECTION keyword to the widget creation routine.

Display Headers

The Display Headers attribute determines if the table headings, the row and column labels, are displayed. By default, this value is set to True, indicating that table heading should be displayed. To disable the display of table headings, set this value to False.

In the generated `*.pro` file, the False value is specified with the NO_HEADERS keyword to the widget creation routine.

Editable

The Editable attribute determines if the table widget is editable or not. By default, this value is set to False, which means the text widget is not editable, and the text is read-only. To make the text widget editable, set this value to True.

In the generated `*.pro` file, this value is specified with the EDITABLE keyword to the widget creation routine.

Number of Columns

The Number of Columns attribute specifies the number of columns in the table widget. This value sets the full, virtual width of the table. By default, it is set to 6.

In the generated `*.pro` file, this value is specified with the XSIZE keyword to the widget creation routine.

Note

To have a scrollable table, set the Scroll attribute to True. Then, to specify the visible size of the table, set the Viewport Columns attribute.

Number of Rows

The Number of Rows attribute specifies the number of rows in the table widget. This value sets the full, virtual height of the table. By default, it is set to 6.

In the generated `*.pro` file, this value is specified with the YSIZE keyword to the widget creation routine.

Note

To have a scrollable table, set the Scroll attribute to True. Then, to specify the visible size of the table, set the Viewport Columns attribute.

Resize Columns

The Resize Columns attribute determines if this user can resize table columns. By default, this value is set to True, indicating that the user can resize the columns. To specify that the columns of the table are not resizeable by the user, set this value to False.

In the generated `*.pro` file, this value is specified with the RESIZEABLE_COLUMNS keyword to the widget creation routine.

Note

If you set the Display Headers attribute to False, the ability to resize the columns is automatically disabled.

Row/Column Major

The Row/Column Major attribute specifies how data is transferred to the table widget, either by Row or by Column. By default, this value is set to Row, indicating that the data should be read into the table as if each element of the vector is a structure containing one row's data. To specify that the data should be read into the table as if each element of the vector is a structure containing one column's data, set this value to Column. Note that for either setting to work properly the structures must all be of the same type, and must have one field for each column or row in the table.

In the generated `*.pro` file, this value is specified with the ROW_MAJOR or the COLUMN_MAJOR keyword to the widget creation routine.

Row Labels

The Row Labels attribute specifies the labels for the table rows. By default, this value is set to empty strings, but you can set it to any set of strings. To set the labels for table columns, use the Column Labels attribute.

To enter more than one string in the Value field, type in a string, then press Control+Enter (at the end of each line). This moves you to the next line, or the next label for a row. When you have entered as many labels as you want, press Enter to set the values.

In the generated `*.pro` file, this value is specified with the ROW_LABELS keyword to the widget creation routine.

Scroll

The Scroll attribute determines if the table widget has scroll bars. By default, this value is set to False, indicating that the table will have no scroll bars. To enable scroll bars, set this value to True. If you set this value to True, you can set the size of the scrollable region with the Viewport Rows and Viewport Columns properties.

In the generated `*.pro` file, this value is specified with the SCROLL keyword to the widget creation routine.

Viewport Columns

The Viewport Columns attribute specifies the number of columns that should be visible in the scroll area of the table widget. By default, this value is set to 6.

If you first set the Scroll attribute to True, you can then set this value to any size from 0 to *n* columns within the limits of your full table size. The full table size, or virtual width in columns, is set with the Number of Columns attribute.

This attribute is used only when the Component Sizing attribute is set to Default. If you set the Component Sizing attribute to Explicit, either through the Properties dialog or by dragging the component to specific size, the Viewport Columns attribute is ignored, and the X Size and the Y Size properties are used.

In the generated `*.pro` file, this value is specified with the X_SCROLL_SIZE keyword to the widget creation routine.

Viewport Rows

The Viewport Rows attribute specifies the number of rows that should be visible in the scroll area of the table widget. By default, this value is set to 6.

If you first set the Scroll attribute to True, you can then set this value to any size from 0 to *n* rows, within the limits of your full table size. The full table size, or virtual height in rows, is set with the Number of Rows attribute.

This attribute is used only when the Component Sizing attribute is set to Default. If you set the Component Sizing attribute to Explicit, either through the Properties dialog or by dragging the component to specific size, the Viewport Rows attribute is ignored, and the X Size and the Y Size properties are used.

In the generated `*.pro` file, this value is specified with the Y_SCROLL_SIZE keyword to the widget creation routine.

Table Widget Events

For table widgets, you can set common event properties and table-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see "Common Events" on page 651.

These are the table widget-specific event properties:

OnCellSelect

The OnCellSelect value is the routine that is called when cells are selected in the table. When you set this value, the calling sequence looks like this in the generated `*_eventcb.pro` file:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when range of cells is selected or deselected and is of the following type:

```
{ WIDGET_TABLE_CELL_SEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:4,
   SEL_LEFT:0L, SEL_TOP:0L, SEL_RIGHT:0L, SEL_BOTTOM:0L }
```

The range of cells selected is given by the zero-based indices into the table specified by the SEL_LEFT, SEL_TOP, SEL_RIGHT, and SEL_BOTTOM fields. When cells are deselected, either by changing the selection or by clicking in the upper left corner of the table, an event is generated in which the SEL_LEFT, SEL_TOP, SEL_RIGHT, and SEL_BOTTOM fields contain the value -1.

Note

Two WIDGET_TABLE_CELL_SEL events are generated when an existing selection is changed to a new selection. If your code uses this event, be sure to differentiate between select and deselect events.

OnColWidth

The OnColWidth value is the routine that is called when the column width is changed. When you set this value, the calling sequence looks like this in the generated `*_eventcb.pro` file:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when a column width is changed by the user and is of the following type:

```
{ WIDGET_TABLE_COLUMN_WIDTH, ID:0L, TOP:0L, HANDLER:0L, TYPE:7,
```

```
   COLUMN:0L, WIDTH:0L }
```

COLUMN contains the zero-based column number, and WIDTH contains the new width.

OnDelete

The OnDelete value is the routine that is called when text is deleted from the table. When you set this value, the calling sequence looks like this in the generated `*_eventcb.pro` file:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when any amount of text is deleted from a cell of a table widget and is of the following type:

```
{ WIDGET_TABLE_DEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:2, OFFSET:0L,
   LENGTH:0L, X:0L, Y:0L }
```

OFFSET is the (zero-based) character position of the first character deleted, and it is the insertion position that will result when the next character is inserted. LENGTH gives the number of characters involved. The X and Y fields give the zero-based address of the cell within the table.

OnFocus

The OnFocus value is the routine that is called when the keyboard focus of the base changes. When you set it, the calling sequence looks like this in the generated `*_eventcb.pro` file:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the returned event structure, which is of the following type:

```
{ WIDGET_KBRD_FOCUS, ID:0L, TOP:0L, HANDLER:0L, ENTER:0 }
```

ENTER returns 1 (one) if the table widget is gaining the keyboard focus, or 0 (zero) if the table widget is losing the keyboard focus.

OnInsertChar

The OnInsertChar value is the routine that is called when text is inserted in the table. When you set this value, the calling sequence looks like this in the generated `*_eventcb.pro` file:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when a single character is typed into a cell of a table widget and is of the following type:

```
{ WIDGET_TABLE_CH, ID:0L, TOP:0L, HANDLER:0L, TYPE:0, OFFSET:0L,
   CH:0B, X:0L, Y:0L }
```

OFFSET is the (zero-based) insertion position that will result after the character is inserted. CH is the ASCII value of the character. The X and Y fields indicate the zero-based address of the cell within the table.

OnInsertString

The OnInsertString value is the routine that is called when text is inserted in the table. When you set this value, the calling sequence looks like this in the generated `*_eventcb.pro` file:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when multiple characters are pasted into a cell and is of the following type:

```
{ WIDGET_TABLE_STR, ID:0L, TOP:0L, HANDLER:0L, TYPE:1, OFFSET:0L,
   STR:'', X:0L, Y:0L }
```

OFFSET is the (zero-based) insertion position that will result after the text is inserted. STR is the string to be inserted. The X and Y fields indicate the zero-based address of the cell within the table.

OnInvalidData

The OnInValidData value is the routine that is called when invalid data is set in a cell. When you set this value, the calling sequence looks like this in the generated `*_eventcb.pro` file:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when the text entered by the user does not pass validation, and the user has finished editing the field (by pressing Tab or Enter). The event structure is of the following type:

```
{ WIDGET_TABLE_INVALID_ENTRY, ID:0L, TOP:0L, HANDLER:0L, TYPE:8,
   STR:'', X:0L, Y:0L }
```

STR contains invalid contents entered by the user as a text string. The X and Y fields contain the cell location.

OnTextSelect

The OnTextSelect value is the routine that is called when text is selected in the table. When you set this value, the calling sequence looks like this in the generated `*_eventcb.pro` file:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when an area of text is selected. The event structure is of the following type:

```
{WIDGET_TABLE_TEXT_SEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:3,
   OFFSET:0L, LENGTH:0L, X:0L, Y:0L}
```

This event announces a change in the insertion point. OFFSET is the (zero-based) character position of the first character to be selected. LENGTH gives the number of characters involved. A LENGTH of zero indicates that the widget has no selection, and that the insertion position is given by OFFSET. The X and Y fields indicate the zero-based address of the cell within the table.

Tab Widget Properties

Tab widgets present a display area on which different “pages” (base widgets and their children) can be displayed by selecting the appropriate tab.

Each tabbed area in a tab widget contains a base widget, to which other controls can be added. When initially created, a tab widget will contain a single tab. To add a tab, right-click on the tabbed portion of the tab widget and select “Add Tab”. The new tab is added at the next logical location in the tab widget. To delete a tab, right-click on the tabbed portion of the tab widget and select “Delete Tab”.

When programming in IDL, you create tab widgets using the WIDGET_TAB function. For more information, see “WIDGET_TAB” in the *IDL Reference Guide* manual.

Tab Widget Attributes

For tab widgets, you can set common attributes and tab-specific attributes. For a list of common attributes, see “Common Attributes” on page 648. In addition, you can modify the attributes of the base widget that makes up the displayed area of each tab.

These are the tab widget-specific attributes:

Multiple Rows

The Multiple Rows attribute determines whether tabs will be displayed in multiple rows if the total length of all the tab titles exceeds the width of the largest tab base widget. By default, this value is set to False, which means that if the total length of all the tab titles exceeds the space available, scroll bars will be added to the tab interface to allow the user to cycle through the tabs. Set this value to True to create tabs in multiple rows if the total length exceeds the available space.

Note

If the Location attribute is set to either Left or Right, the Multiple Rows attribute is automatically set to True.

In the generated `*.pro` file, this value is specified with the MULTILINE keyword to the widget creation routine.

Location

The Location attribute determines the display location of the tabs on the tab widget. Possible values are Top, Bottom, Left and Right. The default value is Top.

In the generated `*.pro` file, this value is specified with the LOCATION keyword to the widget creation routine.

Tab Text

The Tab Text attribute specifies the text label used on the currently-selected tab. By default, this attribute contains the name of the base widget that holds the tab's contents.

In the generated `*.pro` file, this value is specified with the TITLE keyword to the widget base creation routine.

Tab Widget Events

For tab widgets, you can set common event properties and tab-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see "Common Events" on page 651.

These are the tab widget-specific event properties:

OnTabChange

The OnTabChange value is the routine that is called when the currently-selected tab of the tab widget changes. When you set this value, the calling sequence looks like this in the generated `*_eventcb.pro` file:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when a user changes the selected tab and is of the following type:

```
{WIDGET_TAB, ID:0L, TOP:0L, HANDLER:0L, TAB:0L}
```

TAB contains the zero-based index of the tab selected.

Tree Widget Properties

Tree widgets present a hierarchical view that can be used to organize a wide variety of data structures and information.

When you first create a tree widget, it appears in the interface as a blank control, containing no nodes. To add nodes to the tree widget, right-click on the tree widget and select **Edit Tree** from the context menu. See "Using the Tree Editor" on page 626 for details on constructing a a tree hierarchy.

When programming in IDL, you create tree widgets using the WIDGET_TREE function. For more information, see "WIDGET_TREE" in the *IDL Reference Guide* manual.

Tree Widget Attributes

For tree widgets, you can set common attributes and tree-specific attributes. For a list of common attributes, see "Common Attributes" on page 648. These are the tree widget-specific attributes:

Multiple Selection

The Multiple Selection attribute determines whether the tree widget can perform multiple selection operations. Set this property to True to enable multiple selection, which allows the user to select multiple tree nodes by holding down the Control key while selecting with the mouse. By default this property is set to False.

In the generated `*.pro` file, this value is specified with the MULTIPLE keyword to the widget creation routine.

Tree Widget Events

For tree widgets, you can set common event properties and tree-specific event properties. By default, event values are *not* set. For a list of events common to all widgets, see "Common Events" on page 651.

These are the tree widget-specific event properties:

OnContextEvent

The OnContextEvent value is the routine name that is called when the user clicks the right-hand mouse button over the tree widget. In the generated `*_eventcb.pro` file, the event calling sequence looks like this:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when the user clicks the right-hand mouse button and is of the following type:

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

The X and Y fields give the device coordinates at which the event occurred, measured from the upper left corner of the base widget.

OnTreeExpand

The OnTreeExpand value is the routine that is called when a folder expand/collapse event is detected. When you set this value, the calling sequence looks like this in the generated `*_eventcb.pro` file:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when a user changes the selected tab, and is of the following type:

```
{WIDGET_TREE_EXPAND, ID:0L, TOP:0L, HANDLER:0L, TYPE:1, EXPAND:0L}
```

The EXPAND field contains 1 (one) if the folder expanded or 0 (zero) if the folder collapsed.

OnTreeSelect

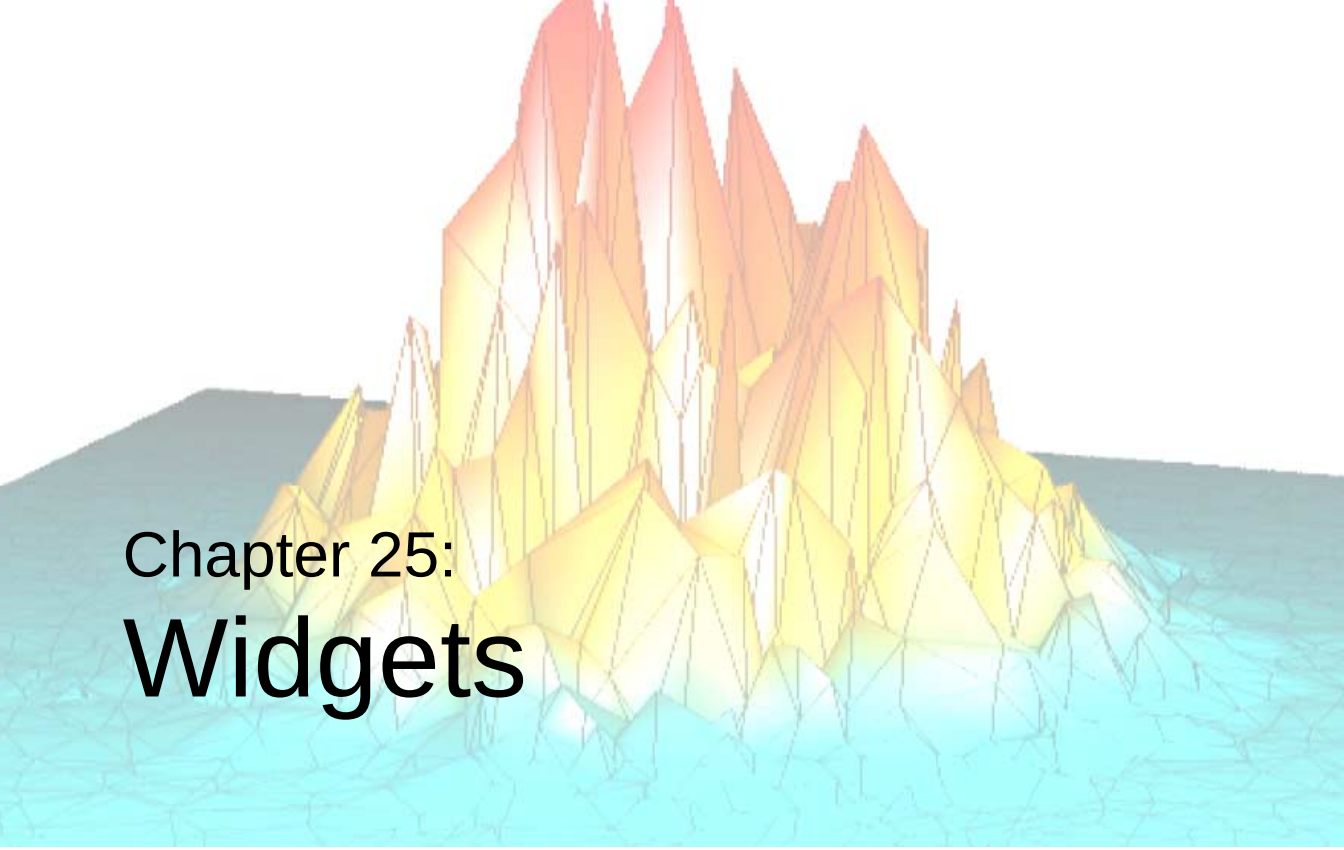
The OnTreeSelect value is the routine that is called when the selection state of an item in the tree changes. When you set this value, the calling sequence looks like this in the generated `*_eventcb.pro` file:

```
pro <RoutineName>, Event
```

where *RoutineName* is the name of the event procedure you specify. `Event` is the event structure returned when a user changes the selected tab, and is of the following type:

```
{WIDGET_TREE_SEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:0, CLICKS:0L}
```

The CLICKS field indicates the number of mouse-button clicks that occurred when the event took place. This field contains 1 (one) when the item is selected, or 2 when the user double-clicks on the item.



Chapter 25: Widgets

The following topics are covered in this chapter:

Overview

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

IDL widgets are significantly easier to use than other alternatives, such as writing a C language program using the native window system 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.

Note

On Microsoft Windows platforms, you can use the IDL GUIBuilder to create user interfaces interactively. The IDL GUIBuilder allows you to create an interface rapidly and generate the IDL source code to create the interface. For information, see Chapter 24, “Using the IDL GUIBuilder”.

Widget Types

IDL supports several types of widgets and widget-like interface elements:

- *Widget primitives* are the base interface elements used to create widget applications. See “Widget Primitives” on page 709 for details.

- *Compound widgets* are more complex interface elements built in the IDL language from the widget primitives. Compound widgets can also be used within widget applications. See “Compound Widgets” on page 722 for details.
- *Dialogs* are widget-like elements that can be called from any IDL application (whether or not it uses other widgets), but which do not belong to a widget hierarchy. Dialogs are useful for informing users of changes in the application state or collecting relatively simple input, such as the answer to a “Yes or No” question or the name of a file. See “Dialogs” on page 731 for details.
- *Utilities* are self-contained widget applications written in the IDL language that can be invoked from the IDL command line or called from within an application. See “Utilities” on page 733 for details.

Widget Programming

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

Because of widget applications’ event-driven nature, creating applications that use widgets is fundamentally different from creating non-widget applications. The widget application model and programming techniques are discussed in Chapter 26, “Creating Widget Applications”.

Events from IDL widgets are generated in the form of an IDL structure variable specific to the widget. Widget events and event-processing are discussed in detail in “Widget Event Processing” in Chapter 26.

Widget Values

Many widget primitives and compound widgets have *widget values* associated with them. Depending on the type of widget, the widget value may represent a static item set by the programmer (the label of a button widget, for example) or a dynamic value set by the user (the numerical value of a slider widget, for example).

Widget values are retrieved from a widget using the GET_VALUE keyword to the WIDGET_CONTROL procedure, and set either when the widget is created or using the SET_VALUE keyword to WIDGET_CONTROL. Descriptions of widget value

data types and default values are included along with the descriptions of individual widgets in the following sections.

Widgets can also have *user values*. A widget's user value is an IDL variable, and can thus be of any of IDL's data types. User values can contain any information the programmer wants to include; they are not examined or used by IDL except as specified by the widget application programmer. User values and their role in widget programming are discussed in "Widget User Values" in Chapter 26.

Instantiating Widgets

When you call a routine that creates a widget, IDL "creates" the widget and assigns it a unique identifier (the *widget ID*). For example, the following IDL statements create a base widget that holds a button widget, and stores the widgets' identifiers in the variables `base` and `button`:

```
base = WIDGET_BASE()
button = WIDGET_BUTTON(base, VALUE='My Button')
```

At this point, the widgets are nothing more than data structures (referred to as *widget records*) in IDL's memory. Nothing appears on screen, and in fact IDL has yet to calculate the sizes of the widgets or the way they will appear.

In order to instantiate the widget — that is, to create the final form of the widget that will be displayed from components supplied by the platform-specific user interface toolkit and (in most cases) make it appear on screen — the widgets must be *realized*. Realization occurs with a call to the WIDGET_CONTROL procedure, using the REALIZE keyword:

```
WIDGET_CONTROL, base, /REALIZE
```

After this command has been issued, the widgets appear on the computer screen.

Realization, and the related concepts of *mapping* and *sensitivity*, are discussed in greater detail in Chapter 26, "Creating Widget Applications". While reading this chapter, it is sufficient to understand that certain operations on widgets, such as retrieving size or other information, can only take place *after* the widget has been realized.

Widget Primitives

Widget primitives are created by functions with names like WIDGET_BASE and WIDGET_BUTTON. IDL provides the following widget primitives:

ActiveX	Base	Button	ComboBox
Draw	Droplist	Label	List
PropertySheet	Slider	Tab	Table
Text	Tree		

The following sections describe each widget primitive, along with its widget value, if any.

ActiveX

An ActiveX widget is a special widget type that is available on Microsoft Windows installations of IDL. An ActiveX widget instantiates an ActiveX control within an IDL widget hierarchy. ActiveX widgets are controlled using the object methods of the underlying ActiveX control rather than via the WIDGET_CONTROL and WIDGET_INFO routines.

ActiveX widgets are created using the WIDGET_ACTIVEX function. See "WIDGET_ACTIVEX" in the *IDL Reference Guide* and Chapter 5, "Using ActiveX Controls in IDL" in the *External Development Guide* manual for more information.

ActiveX Widget Values

The widget value of an ActiveX widget is an object reference to the IDLcomActiveX object created to hold the ActiveX control. The value cannot be set, and cannot be retrieved before the widget is realized.

ActiveX Widget Events

ActiveX widgets return a basic widget event structure augmented by fields that are specific to the ActiveX control. See "ActiveX Widget Events" in Chapter 5 of the *External Development Guide* manual for details.

Base

A base is a widget used to hold other widgets, including other base widgets. Base widgets can optionally contain scroll bars that allow the base to be larger than the space on the screen. In this case, only part of the base is visible at any given time, and the scroll bars are used to control which part is visible.

Base widgets are created by the WIDGET_BASE function. See "WIDGET_BASE" in the *IDL Reference Guide* for more information.

Top-level bases are a special class of base widget created without a parent widget ID. Every widget hierarchy has exactly one top-level base. (Multiple widget hierarchies, represented by their top-level bases, can be organized into an application hierarchy by specifying the GROUP_LEADER keyword. See "Using Multiple Widget Hierarchies" on page 787 for additional discussion of widget applications with multiple widget hierarchies.)

Base Widget Values

The base widget does not have a widget value.

Base Widget Events

Base widgets do not generate any events by default. They can be configured to generate events when the base widget receives the keyboard focus or the user clicks the right-hand mouse button. In addition, top-level bases can be configured to generate events when the base is resized, moved, iconified, or killed. See "Events Returned by Base Widgets" under "WIDGET_BASE" in the *IDL Reference Guide* for details.

Button

A button widget is a pushbutton that is activated by moving the mouse cursor over the widget button and pressing a mouse button. Button widgets are created by the WIDGET_BUTTON function. See "WIDGET_BUTTON" in the *IDL Reference Guide* for more information.

Figure 25-1: A button widget.

Button Widget Values

The widget value of a button widget is a scalar string (text) or byte array (bitmap) that represents the button's label. See "Using Button Widgets" in Chapter 27 for additional details on using bitmap values for button widgets.

Button Widget Events

Button widgets generate an event when a user clicks the mouse-pointer on the button. See "Events Returned by Button Widgets" under "WIDGET_BUTTON" in the *IDL Reference Guide* for details.

ComboBox

ComboBox widgets display a single entry from a list of options. When selected, they reveal the entire list. When a new option is selected from this list, the list disappears and the new selection is displayed. The main difference between the ComboBox widget and the droplist widget is that the text field of the ComboBox can be made editable, allowing the user to enter of a value that is not on the list.

ComboBox widgets are created by the WIDGET_COMBOBOX function. See "WIDGET_COMBOBOX" in the *IDL Reference Guide* for more information.

Figure 25-2: A ComboBox widget

Combobox Widget Values

The widget value of a ComboBox widget is a scalar string or string array representing the list elements.

Combobox Widget Events

Combobox widgets generate an event when the user selects or edits a value from the drop-down list. See "Widget Events Returned by Combobox Widgets" under "WIDGET_COMBOBOX" in the *IDL Reference Guide* for details.

Draw

Draw widgets offer a rectangular area that works like a standard IDL graphics window. Draw widgets can use either Direct graphics or Object graphics, depending on how they are created. Any graphical output that can be produced by IDL can be directed to one of these widgets, either through the WSET function or by using the object reference of a draw widget's IDLgrWindow object. Draw widgets can optionally contain scrollbars that allow examining a graphical region larger than the area displayed by the widget. Draw widgets are created by the WIDGET_DRAW function. See "Using Draw Widgets" on page 815 and "WIDGET_DRAW" in the *IDL Reference Guide* for more information.

Figure 25-3: A draw widget.

Draw Widget Values

For draw widgets created using Direct Graphics, the widget value is an integer representing the IDL window number for use with direct graphics routines, such as WSET. For draw widgets created using Object Graphics, the widget value is an object reference to the IDLgrWindow object. This value cannot be set or modified.

Draw Widget Events

Draw widgets do not generate any events by default. They can be configured to generate events when a mouse button is pressed or released over the widget, when the mouse cursor moves over the widget, when the draw widget is obscured by another window on the screen or revealed, or when the user presses a keyboard key. See

"Widget Events Returned by Draw Widgets" under "WIDGET_DRAW" in the *IDL Reference Guide* for details.

Droplist

Droplist widgets display a single entry from a list of options. When selected, they reveal the entire list. When a new option is selected from this list, the list disappears and the new selection is displayed. The main difference between the droplist widget and the ComboBox widget is that the text field of the droplist cannot be made editable. Droplist widgets are created by the WIDGET_DROPLIST function. See "WIDGET_DROPLIST" in the *IDL Reference Guide* for more information.

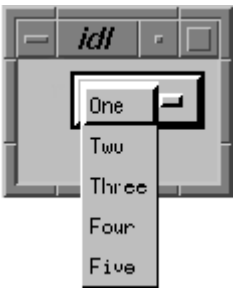


Figure 25-4: A droplist widget

Note

The appearance of the droplist widget differs on Motif and Windows platforms. Under Windows, the droplist widget looks the same as the ComboBox widget; under Motif, it appears as shown in the figure above.

Droplist Widget Values

The widget value of a droplist widget is a scalar string or string array representing the list elements. This value can only be set; it cannot be retrieved.

Droplist Widget Events

Droplist widgets generate an event when the user selects a value from the drop-down list. See "Widget Events Returned by Droplist Widgets" under "WIDGET_DROPLIST" in the *IDL Reference Guide* for details.

Label

Label widgets display static text. They are similar to single-line text widgets but are optimized for small labeling purposes. If you need to display more than a single line of text, or if the text must be editable by the user, use a text widget. Label widgets are created by the WIDGET_LABEL function. See "WIDGET_LABEL" in the *IDL Reference Guide* for more information.

Figure 25-5: A label widget.

Label Widget Values

The widget value of a label widget is a scalar string representing the label text.

Label Widget Events

Label widgets do not generate any events.

List

A list widget offers the user a list of text elements from which to choose. Users can select an item by pointing with the mouse cursor and pressing a button. List widgets have a vertical scrollbar when there are more list items than are specified by the HEIGHT keyword. List widgets are created by the WIDGET_LIST function. See "WIDGET_LIST" in the *IDL Reference Guide* for more information.

Figure 25-6: A list widget.

List Widget Values

The widget value of a list widget is a scalar string or string array representing the list elements. This value can only be set; it cannot be retrieved.

List Widget Events

List widgets generate an event when the user selects a value or values from the list. They can also be configured to generate events when the user clicks the right-hand mouse button over the widget. See "Widget Events Returned by List Widgets" under "WIDGET_LIST" in the *IDL Reference Guide* for details.

PropertySheet

PropertySheet widgets enable the user to view and edit the properties of an object subclassed from the IDLitComponent class. (All IDLgr* and IDLit* objects are subclassed from IDLitComponent.) The name of the changed property is placed into an IDL event, and the object is updated when this event is processed. An existing property sheet can also be assigned a new component, causing it to reload with the new list of properties and their values.

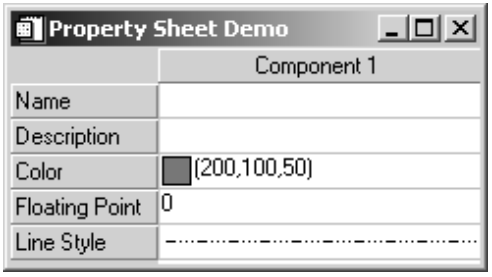


Figure 25-7: A Property Sheet Widget

Property Sheet Widget Data Types and Controls

The following controls are available for the corresponding data types in property sheet widgets:

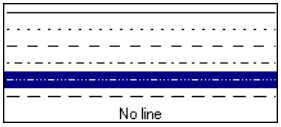
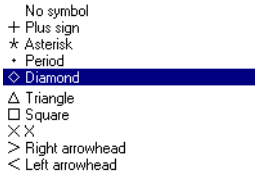
Data Type	Control
Boolean	Droplist with two choices: True or False
Number	One of the following: • Edit control that accepts +, -, 0123456789, plus decimal points, e, E, d and D for floating point numbers. • Slider
String	Text box, expandable when string length exceeds visible area.
Color	Color picker.
Line Style	Droplist displaying seven line style choices: 
Line Thickness	Droplist displaying ten line thickness choices: 
Symbol	Droplist displaying eight symbol choices: 

Table 25-1: WIDGET_PROPERTYSHEET Controls for Data Types

Data Type	Control
String list	Droplist displaying a list of strings.
User-Defined	**Edit** button linked to user-defined control (see “User-defined Properties” on page 830).

Table 25-1: WIDGET_PROPERTYSHEET Controls for Data Types

Property Sheet Widget Values

The property sheet widget's value is an object reference (or array of object references). The object must be a subclass of IDLitComponent. Whenever the value is set, via WIDGET_CONTROL's SET_VALUE keyword, the property sheet widget is loaded with the registered, visible properties of the new component. Setting the value to a null object will clear the property sheet widget. A null object can be created by calling the OBJ_NEW function without any arguments:

```
nullObject = OBJ_NEW()
```

Property Sheet Widget Events

Property sheet widgets generate events whenever propertysheet contents are modified. See “Widget Events Returned by Property Sheet Widgets” under “WIDGET_PROPERTYSHEET” in the *IDL Reference Guide* for details.

Slider

Slider widgets are used to select or indicate a value within a range of possible integer values. They consist of a rectangular region that represents the possible range of values. Inside this region is a sliding pointer that displays the current value. This pointer can be manipulated by the user via the mouse or from within IDL by the WIDGET_CONTROL procedure. Slider widgets are created by the WIDGET_SLIDER function. See “WIDGET_SLIDER” in the *IDL Reference Guide*

for more information. Sliders that indicate a floating-point value can be created using the CW_FSLIDER compound widget.

Figure 25-8: A slider widget.

Slider Widget Values

The widget value of a slider widget is the integer value of the current slider position.

Slider Widget Events

Slider widgets generate events when the mouse is used to change their value. See "Widget Events Returned by Slider Widgets" under "WIDGET_LIST" in the *IDL Reference Guide* for details.

Tab

Tab widgets create a "tabbed" interface that allows the user to select one of a list of rectangular display areas to be displayed in a single space. The displayed interface elements are contained in base widgets — that is, selecting a tab displays the contents of a specified base widget within the tab widget. See "Using Tab Widgets" on page 859 and "WIDGET_TAB" in the *IDL Reference Guide* for more information.

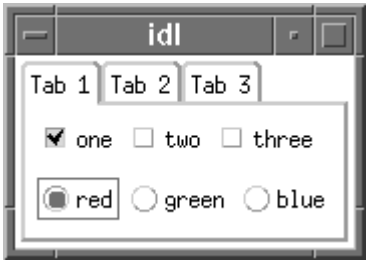


Figure 25-9: A tab widget.

Tab Widget Values

The tab widget does not have a widget value.

Tab Widget Events

Tab widgets generate an event when a new tab is selected. See “Widget Events Returned by Tab Widgets” under “WIDGET_TAB” in the *IDL Reference Guide* for details.

Table

Table widgets are used to display information in tabular format. Individual table cells (or ranges of cells) can be selected for editing by the user. Table widgets are created by the WIDGET_TABLE function. See “Using Table Widgets” on page 848 and “WIDGET_TABLE” in the *IDL Reference Guide* for more information.

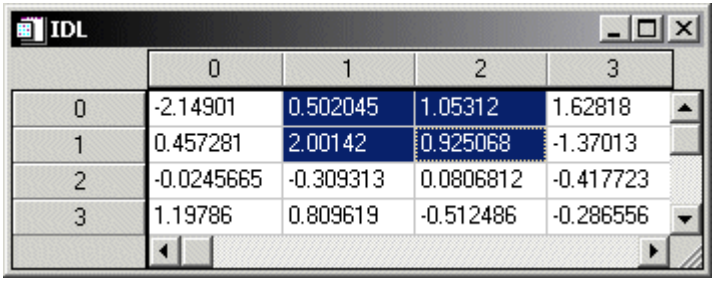


Figure 25-10: A table widget.

Table Widget Values

The widget value of a table widget can be either a two-dimensional array of any data type or a vector of structures, representing the contents of the table. If the table data is supplied as a vector of structures, the structures must contain one field for each column (if the COLUMN_MAJOR keyword to WIDGET_TABLE is set) or one field for each row (if the ROW_MAJOR keyword to WIDGET_TABLE is set, or if neither keyword is set). The individual structure fields can be of any data type.

Table Widget Events

Table widgets generate events when table cell contents are selected, deselected, or modified, or when table rows or columns are resized. Table widgets can also be

configured to generate events when the keyboard focus changes. See "Widget Events Returned by Table Widgets" under "WIDGET_TABLE" in the *IDL Reference Guide* for details.

Text

Text widgets are used to display text and to get text input from the user. They can have one or more lines and can optionally contain scroll bars that allow viewing more text than can otherwise be displayed. Text widgets are created by the WIDGET_TEXT function. See "WIDGET_TEXT" in the *IDL Reference Guide* for more information.

Figure 25-11: A text widget.

Text Widget Values

The widget value of a text widget is a scalar string or string array representing the contents of the text widget. When setting this value using WIDGET_CONTROL, by default the old text is replaced by the new text. Specifying the APPEND keyword to WIDGET_CONTROL causes the new text to be appended to the old text.

Text Widget Events

Text widgets generate events when the contents of the text widget change. They can also be configured to generate events when the keyboard focus changes and when the user clicks the right-hand mouse button on the widget. See "Widget Events Returned by Text Widgets" under "WIDGET_TABLE" in the *IDL Reference Guide* for details.

Tree

Tree widgets are used to display a hierarchical view of a complex data structure. Branches on the tree can be expanded or collapsed by the user, displaying different

portions of the structure. See “Using Tree Widgets” on page 868 and “WIDGET_TREE” in the *IDL Reference Guide* for more information.

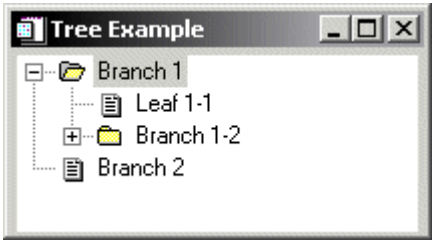


Figure 25-12: A tree widget.

Tree Widget Values

The widget value of a tree widget is a string that represents the text displayed by the branch or leaf of the tree. If the widget value is not explicitly set, either in the call to WIDGET_TREE or using the SET_VALUE keyword to WIDGET_CONTROL, the default value of “Tree” is provided.

Tree Widget Events

Tree widgets generate events when tree elements are selected or deselected, expanded or collapsed. They can also be configured to generate events when the user clicks the right-hand mouse button over the widget. See “Widget Events Returned by Tree Widgets” under “WIDGET_TREE” in the *IDL Reference Guide* for details.

Compound Widgets

A compound widget is a complete, self-contained, reusable widget sub-tree that behaves to a large degree just like a widget primitive, but which is written in the IDL language. Compound widgets allow the development of reusable widget code, much like a GUI subroutine.

Widget Values of Compound Widgets

Many compound widgets have associated values. Initial values can often be specified using the VALUE keyword to the creation routine. Note, however, that in some cases widget values of compound widgets cannot be set until after the widget is realized; values are thus set, obtained, or changed using the GET_VALUE and SET_VALUE keywords to the WIDGET_CONTROL procedure. See the documentation for the individual compound widget creation routines in the *IDL Reference Guide* for more detailed information.

Compound Widgets Provided with IDL

Compound widget routines provided with IDL can be found (along with many other routines that use the widgets) in the `lib` subdirectory of the IDL distribution. All RSI-supplied compound widget filenames begin with "CW_" to make them easier to identify. The following compound widgets are included in the IDL distribution.

CW_ANIMATE	CW_ARCBALL	CW_BGROUP
CW_CLR_INDEX	CW_COLORSEL	CW_DEFROI
CW_FIELD	CW_FILESEL	CW_FORM
CW_FSLIDER	CW_LIGHT_EDITOR	CW_ORIENT
CW_PALETTE_EDITOR	CW_PDMENU	CW_RGBSLIDER
CW_ZOOM		

See "Compound Widgets" on page 767 for information on writing your own compound widgets.

Compound Widget Categories

The compound widgets included with IDL fall into the following functional categories:

Animation

CW_ANIMATE

Color Manipulation

CW_CLR_INDEX

CW_COLORSEL

CW_PALETTE_EDITOR

CW_RGBSLIDER

Data Entry and Display

CW_FIELD

CW_FILESEL

CW_FORM

Image Manipulation

CW_DEFROI

CW_LIGHT_EDITOR

CW_ZOOM

Orientation

CW_ARCBALL

CW_ORIENT

User Interface

CW_BGROUP

CW_FSLIDER

CW_PDMENU

CW_ANIMATE

The CW_ANIMATE compound widget — along with its associated routines — displays an animated sequence of images. See "CW_ANIMATE" in the *IDL Reference Guide*.

Note

Three routines associated with the CW_ANIMATE compound widget — CW_ANIMATE_GETP, CW_ANIMATE_LOAD, and CW_ANIMATE_RUN — do not create compound widgets themselves, but act on an existing CW_ANIMATE widget.

CW_ANIMATE Widget Value

The CW_ANIMATE compound widget does not have a widget value.

CW_ANIMATE Widget Events

CW_ANIMATE generates an event when the user presses the Done button. See "Widget Events Returned by the CW_ANIMATE Widget" in the *IDL Reference Guide* for details.

CW_ARCBALL

The CW_ARCBALL compound widget allows the user to intuitively specify three-dimensional orientations. See "CW_ARCBALL" in the *IDL Reference Guide*.

CW_ARCBALL Widget Value

The widget value of a CW_ARCBALL compound widget is a 3 by 3 array containing the three-dimensional rotation matrix.

CW_ARCBALL Widget Events

CW_ARCBALL generates an event containing a 3 by 3 array containing the three-dimensional rotation matrix. See "Widget Events Returned by the CW_ARCBALL Widget" in the *IDL Reference Guide* for details.

CW_BGROUP

The CW_BGROUP compound widget simplifies creation of a cluster of buttons. Button groups can be simple menus in which each button acts independently, *exclusive* groups (also known as "radio buttons"), or *non-exclusive* groups (often called "checkboxes"). See "CW_BGROUP" in the *IDL Reference Guide*.

CW_BGROUP Widget Value

For "normal" button groups, a CW_BGROUP compound widget does not have a widget value. For exclusive button groups, the widget value is the integer index of the

selected button. For non-exclusive button groups, the value is a vector indicating which buttons are selected. The initial value of a button group can be set using the SET_VALUE keyword to CW_BGROUP or using WIDGET_CONTROL.

CW_BGROUP Widget Events

CW_BGROUP generates an event that specifies which button or buttons were selected. See "Widget Events Returned by the CW_BGROUP Widget" in the *IDL Reference Guide* for details.

CW_CLR_INDEX

The CW_CLR_INDEX compound widget displays a color bar and allows the user to select a color index. See "CW_CLR_INDEX" in the *IDL Reference Guide*.

CW_CLR_INDEX Widget Value

The widget value of a CW_CLR_INDEX compound widget is the index of the color selected. The value cannot be set before the widget is realized.

CW_CLR_INDEX Widget Events

CW_CLR_INDEX generates an event that specifies the selected color index. See "Widget Events Returned by the CW_CLR_INDEX Widget" in the *IDL Reference Guide* for details.

CW_COLORSEL

The CW_COLORSEL compound widget displays all the colors in the current colormap and allows the user to select color indices. See "CW_COLORSEL" in the *IDL Reference Guide*.

CW_COLORSEL Widget Value

The widget value of a CW_COLORSEL compound widget is the index of the color selected. The value cannot be set before the widget is realized.

CW_COLORSEL Widget Events

CW_COLORSEL generates an event that specifies the selected color index. See "Widget Events Returned by the CW_COLORSEL Widget" in the *IDL Reference Guide* for details.

CW_DEFROI

The CW_DEFROI compound widget allows you to specify a *region of interest* within a draw widget. See "CW_DEFROI" in the *IDL Reference Guide*.

CW_DEFROI Widget Value

The CW_DEFROI compound widget does not have a widget value.

CW_DEFROI Widget Events

CW_DEFROI does not return any events.

CW_FIELD

The CW_FIELD compound widget simplifies building data-entry interfaces by combining label and text widgets. See "CW_FIELD" in the *IDL Reference Guide*.

CW_FIELD Widget Value

The widget value of a CW_FIELD compound widget is the string value of the text portion of the field widget.

CW_FIELD Widget Events

CW_FIELD returns an event that specifies the current value of the field, the type of data, and whether the data has been updated. See "Widget Events Returned by the CW_FIELD Widget" in the *IDL Reference Guide* for details.

CW_FILESEL

The CW_FILESEL compound widget allows you to select a file. See "CW_FILESEL" in the *IDL Reference Guide*.

CW_FILESEL Widget Value

The CW_FILESEL compound widget does not have a widget value.

CW_FILESEL Widget Events

CW_FILESEL generates an event that specifies the name of the selected file, whether the user completed the file selection operation, and the filename filter used. See "Widget Events Returned by CW_FILESEL" in the *IDL Reference Guide* for details.

CW_FORM

The CW_FORM compound widget allows you to create simple forms with text, numeric fields, buttons, and droplists. See "CW_FORM" in the *IDL Reference Guide*.

CW_FORM Widget Value

The widget value of the CW_FORM compound widget is a structure of tag/value pairs for each field in the form. The value cannot be set before the widget is realized.

CW_FORM Widget Events

CW_FORM generates an event that specifies the which field within the form changed and the new value. See "Widget Events Returned by the CW_FORM Widget" in the *IDL Reference Guide* for details.

CW_FSLIDER

The CW_FSLIDER compound widget is a version of the slider widget that handles floating-point values. See "CW_FSLIDER" in the *IDL Reference Guide*.

CW_FSLIDER Widget Value

The widget value of the CW_FSLIDER compound widget is the floating-point numeric value of the slider.

CW_FSLIDER Widget Events

CW_FSLIDER generates an event that specifies the current value of the slider and a flag specifying whether events are generated as the slider is dragged. See "Widget Events Returned by the CW_FSLIDER Widget" in the *IDL Reference Guide* for details.

CW_LIGHT_EDITOR

The CW_LIGHT_EDITOR compound widget allows you to edit properties of existing IDLgrLight objects in a view. See "CW_LIGHT_EDITOR" in the *IDL Reference Guide*.

Note

Two routines associated with the CW_LIGHT_EDITOR compound widget — CW_LIGHT_EDITOR_GET and CW_LIGHT_EDITOR_SET — do not create

compound widgets themselves, but act on an existing CW_LIGHT_EDITOR widget.

CW_LIGHT_EDITOR Widget Value

The CW_LIGHT_EDITOR compound widget does not have a widget value.

CW_LIGHT_EDITOR Widget Events

CW_LIGHT_EDITOR generates events when the light is selected and when a light is modified. See "Widget Events Returned by the CW_LIGHT_EDITOR Widget" in the *IDL Reference Guide* for details.

CW_ORIENT

The CW_ORIENT compound widget allows the user to interactively adjust the three-dimensional drawing transformation. SEE "CW_ORIENT" in the *IDL Reference Guide*.

CW_ORIENT Widget Value

The CW_ORIENT compound widget does not have a widget value.

CW_ORIENT Widget Events

CW_ORIENT generates a standard event (containing only the three standard fields) when the three dimensional drawing transformation has been altered. See "Widget Events Returned by the CW_ORIENT Widget" in the *IDL Reference Guide* for details.

CW_PALETTE_EDITOR

The CW_PALETTE_EDITOR compound widget creates allows you to display and edit color palettes. SEE "CW_PALETTE_EDITOR" in the *IDL Reference Guide*.

Note

Two routines associated with the CW_PALETTE_EDITOR compound widget — CW_PALETTE_EDITOR_GET and CW_PALETTE_EDITOR_SET — do not create compound widgets themselves, but act on an existing CW_PALETTE_EDITOR widget.

CW_PALETTE_EDITOR Widget Value

The CW_PALETTE_EDITOR compound widget does not have a widget value.

CW_PALETTE_EDITOR Widget Events

CW_PALETTE_EDITOR generates events when the selection region is changed and when the color palette has been modified. See "Widget Events Returned by the CW_PALETTE_EDITOR Widget" in the *IDL Reference Guide* for details.

CW_PDMENU

The CW_PDMENU compound widget creates pulldown menus, which can include sub-menus, from a set of buttons. See "CW_PDMENU" in the *IDL Reference Guide*.

CW_PDMENU Widget Value

The CW_PDMENU compound widget does not have a widget value.

CW_PDMENU Widget Events

CW_PDMENU generates an event that specifies the index, widget ID, or name of the menu item selected. See "Widget Events Returned by the CW_PDMENU Widget" in the *IDL Reference Guide* for details.

CW_RGBSLIDER

The CW_RGBSLIDER compound widget allows the user to adjust color values using the RGB, CMY, HSV, and HLS color systems. See "CW_RGBSLIDER" in the *IDL Reference Guide*.

CW_RGBSLIDER Widget Value

The widget value of the CW_RGBSLIDER compound widget is a 3-element [*r, g, b*] vector representing the RGB value selected.

CW_RGBSLIDER Widget Events

CW_RGBSLIDER generates an event that specifies the red, green, and blue values selected. See "Widget Events Returned by the CW_RGBSLIDER Widget" in the *IDL Reference Guide* for details.

CW_ZOOM

The CW_ZOOM compound widget displays original and zoomed images side-by-side. See "CW_ZOOM" in the *IDL Reference Guide*.

CW_ZOOM Widget Value

The initial widget value of the CW_ZOOM compound widget is a byte array that represents the full image to be displayed in the widget. The value cannot be set before the widget is realized.

After the image has been zoomed, the widget value is a byte array that represents the magnified area of the image.

CW_ZOOM Widget Events

CW_ZOOM generates an event that specifies the dimensions of the zoomed image and the corresponding coordinates within the original image. See "Widget Events Returned by the CW_ZOOM Widget" in the *IDL Reference Guide* for details.

Dialogs

A dialog is a widget-like user interface element that is not part of a widget hierarchy. Dialogs are used to report errors, prompt for a user response, or collect small amounts of user input such as the answer to a yes-or-no question or the name of a file. They have short lifetimes, and disappear after serving their purpose.

Dialogs are *modal* (or “blocking”), which means that when a dialog is displayed, no other interface elements (widgets or compound widgets) can be manipulated until the user dismisses the dialog. While the dialog is not part of any widget hierarchy, you can specify a widget over which the dialog will be centered on screen, making it possible to visually associate the dialog with a specific widget application.

IDL supplies the following dialogs:

DIALOG_MESSAGE	DIALOG_PICKFILE
DIALOG_PRINTERSETUP	DIALOG_PRINTJOB
DIALOG_READ_IMAGE	DIALOG_WRITE_IMAGE

DIALOG_MESSAGE

The DIALOG_MESSAGE function displays a warning, informational message, or error message, and blocks manipulation of other IDL widgets until the user dismisses the dialog by clicking one of its buttons. See “DIALOG_MESSAGE” in the *IDL Reference Guide* for details.

DIALOG_PICKFILE

The DIALOG_PICKFILE function allows you to choose a file or directory via a graphical interface, and returns a string containing the name of the selected file or directory. See “DIALOG_PICKFILE” in the *IDL Reference Guide* for more information.

DIALOG_PRINTERSETUP

The DIALOG_PRINTERSETUP function opens an operating system-native dialog for setting the applicable properties for a particular printer. See “DIALOG_PRINTERSETUP” in the *IDL Reference Guide* for more information.

DIALOG_PRINTJOB

The DIALOG_PRINTJOB function opens an operating system-native dialog that allows you to set the parameters for a printing job (such as the number of copies to print). See "DIALOG_PRINTJOB" in the *IDL Reference Guide* for more information.

DIALOG_READ_IMAGE

The DIALOG_READ_IMAGE function provides a graphical interface allowing the user to select an image file and read it into an IDL variable. A preview of the selected image is provided. See "DIALOG_READ_IMAGE" in the *IDL Reference Guide* for more information.

DIALOG_WRITE_IMAGE

The DIALOG_WRITE_IMAGE function provides a graphical interface allowing the user to save an IDL array variable as an image file, selecting a location and image file type. See "DIALOG_WRITE_IMAGE" in the *IDL Reference Guide* for more information.

Utilities

IDL provides a number of stand-alone “utilities” written in the IDL language. The utility routines are widget applications that perform relatively complex operations; they are designed to be used either independently or as part of a larger widget application. Names of utility routines are prefaced with the letter “X”.

Utility routines can be called from any IDL application or from the IDL command line. Although utility routines cannot be inserted directly into a widget application (becoming part of the application’s widget hierarchy), they can be linked to a widget application in such a way (via the GROUP keyword) that when the widget application is iconized or destroyed, the utility is iconized or destroyed as well. Utility routines can also be configured as *modal* applications, requiring that the user exit from the utility before returning to the widget application that called it. See “Using Multiple Widget Hierarchies” in Chapter 27 for further discussion of grouping and modal behaviors.

The following utility routines are included with IDL:

XBM_EDIT	XDISPLAYFILE	XDXF
XFONT	XINTERANIMATE	XLOADCT
XMTOOL	XOBJVIEW	XPALETTE
XPCOLOR	XPLOT3D	XROI
XSURFACE	XVAREDIT	XVOLUME

XBM_EDIT

The XBM_EDIT utility is a graphical bitmap editor that allows you to create bitmap arrays for use as labels for button widgets. Array definitions can be saved either as array definition text files, for inclusion in IDL code, or as bitmap array data files, which can be reopened in the XBM_EDIT utility. See “XBM_EDIT” in the *IDL Reference Guide* for details.

XDISPLAYFILE

The XDISPLAYFILE utility allows you to display text files using a simple widget interface. In addition to allowing users to read and optionally edit text files, the utility allows you programmatically interact with the text widget. See “XDISPLAYFILE” in the *IDL Reference Guide* for details.

XDXF

The XDXF utility displays a DXF file in an XOBJVIEW viewer, and also displays a dialog that contains block and layer information for the DXF file, allowing the user to turn on and off the display of individual layers. See "XOBJVIEW" below and "XDXF" in the *IDL Reference Guide* for details.

XFONT

The XFONT utility is a UNIX-only widget application that allows you to select an X Windows font name, optionally resetting the current font to the selected font. See "XFONT" in the *IDL Reference Guide* for details.

XINTERANIMATE

The XINTERANIMATE utility displays an animated sequence of images in a simple widget interface using off-screen pixmaps or memory buffers. The speed and direction of the display can be adjusted. See "XINTERANIMATE" in the *IDL Reference Guide* for details.

XLOADCT

The XLOADCT utility provides a simple widget interface to the LOADCT procedure. XLOADCT displays the current Direct Graphics colortable and shows a list of available predefined color tables; clicking on the name of a color table causes that color table to be loaded. See "XLOADCT" in the *IDL Reference Guide* for details.

XMTOOL

The XMTOOL utility displays a simple widget interface for viewing widgets currently being managed by the XMANAGER. See "XMTOOL" in the *IDL Reference Guide* for details.

XOBJVIEW

The XOBJVIEW utility is a complex widget interface that allows you to quickly and easily view and manipulate IDL Object Graphics on screen. It displays objects in an IDL widget with toolbar buttons and menus providing functionality for manipulating, printing, and exporting the resulting graphic. The mouse can be used to rotate, scale,

or translate the overall model shown in a view, or to select graphic objects in the view. See "XOBJVIEW" in the *IDL Reference Guide* for details.

Note

Two routines associated with the XOBJVIEW utility — XOBJVIEW_ROTATE and XOBJVIEW_WRITE_IMAGE — do not provide a graphical interface themselves, but act on an existing instance of the XOBJVIEW utility.

XPALETTE

The XPALETTE utility displays a widget interface that allows interactive creation and modification of colortables using the RGB, CMY, HSV, or HLS color systems. See "XPALETTE" in the *IDL Reference Guide* for details.

XPCOLOR

The XPCOLOR utility displays a simple widget interface that allows you to adjust the value of the current Direct Graphics plotting color (foreground) using sliders, and store the desired color in the global system variable, !P.COLOR. See "XPCOLOR" in the *IDL Reference Guide* for details.

XPLOT3D

The XPLOT3D utility displays a 3D plot in an XOBJVIEW viewer. See "XOBJVIEW" above and "XPLOT3D" in the *IDL Reference Guide* for details.

XROI

The XROI utility allows you to interactively define regions of interest (ROIs) in a specified image file, and to obtain geometry and statistical data about these ROIs. See "XROI" in the *IDL Reference Guide* for details.

XSURFACE

The XSURFACE utility provides a simple graphical interface to the SURFACE and SHADE_SURF commands. Because XSURFACE relies on IDL direct graphics rather than object graphics, the image manipulation facilities are less sophisticated than those of XOBJVIEW. See "XOBJVIEW" above and "XSURFACE" in the *IDL Reference Guide* for details.

XVAREDIT

The XVAREDIT utility provides a simple widget-based editor for the value of any IDL variable. See "XVAREDIT" in the *IDL Reference Guide* for details.

XVOLUME

The XVOLUME utility provides a complex widget interface for viewing and interactively manipulating volume and isosurface data. See "XVOLUME" in the *IDL Reference Guide* for details.

Note

Two routines associated with the XVOLUME utility — XVOLUME_ROTATE and XVOLUME_WRITE_IMAGE — do not provide a graphical interface themselves, but act on an existing instance of the XVOLUME utility.

Chapter 26: Creating Widget Applications

The following topics are covered in this chapter:

About Widget Applications

The flow of control in a widget application is fundamentally different than in other IDL programs. A program written to be used from the IDL command line generally accepts its inputs when the program is invoked. The program then proceeds in a well-defined order to process those inputs and provide some output — a calculated value, a plot, an image, *etc*. In contrast, widget applications are *event driven*. When initially invoked, a widget application displays its user interface and waits for user input, performing actions in the order specified by the user at runtime, rather than in the order determined by the programmer.

This chapter discusses topics related to creating widget user interfaces, controlling widgets, processing events generated by user interaction, and managing the application state of a widget application. Chapter 27, "Widget Application Techniques" explores the use of specific types of widgets in widget applications and discusses methods for creating specific types of interfaces and applications.

Running the Example Code

The example code used in this chapter and in Chapter 27, "Widget Application Techniques" is part of the IDL distribution. All of the examples developed in the text of these chapters are included as `.pro` files in the `examples/widgets` 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" in the *IDL Reference Guide* manual for information on IDL's path.

Other Examples of Widget Programming

In addition to the examples developed in this and the following chapter, a number of simple examples of widget programming can be seen by running the IDL program `wexmaster.pro`, located in the `/examples/widgets/wexmast` folder of the IDL distribution. A widget interface with a pulldown menu of small widget applications should appear.

Widget Programming Concepts

This section discusses some basic ideas and concepts that are central to the process of writing IDL widget applications.

Widget IDs

IDL widgets are uniquely identified via their *widget IDs*. The widget ID is a long integer assigned to the widget when it is first created; this integer is returned as the value of the widget creation function. For example, you might create a base widget with the following IDL command:

```
base = WIDGET_BASE()
```

Here, the IDL variable `base` receives the widget ID of the newly-created top-level base widget.

Routines within your widget application that need to retrieve data from widgets or change their appearance need access to the widgets' IDs. Techniques for passing widget IDs between independent routines in your widget application are discussed in "Working With Widget IDs" on page 752.

Widget Parent/Child Relationships

With one exception (described below), when you create a new widget using one of the WIDGET_* functions, you specify the widget ID of the new widget's *parent widget*. This parent-child relationship defines a *widget hierarchy.*

For example, suppose you have created a base widget whose widget ID is contained in the IDL variable `base`. The following IDL command creates a button widget that is a *child* of the base widget whose widget ID is stored in the variable `base`:

```
button1 = WIDGET_BUTTON(base, VALUE='Test button')
```

In addition to being below `base` in the widget hierarchy, `button1` appears inside `base1` when the base widget is realized on the screen.

The exception to this parent-child rule is a special instance of a base widget called a *top-level base*. A top-level base is different from an "ordinary" base widget in the following ways:

- It does not have a parent widget.
- It serves as the top of a widget hierarchy

- Its widget ID is included in the `TOP` field of every widget event structure generated by other widgets in its hierarchy.

In practice, a widget application always begins with a top-level base. The fact that the widget ID of the top-level base widget is always available in the event structure of widget events is very useful for managing the state of a widget application. This topic is discussed in depth in "Managing Application State" on page 763.

Instantiating and Displaying Widgets

IDL widgets provide a platform-independent abstraction layer on top of the graphical interface toolkits of the different operating systems supported by IDL. This abstraction layer allows you to create a graphical user interface once, and let IDL do the work of translating and displaying the interface on different platforms.

When you use one of the widget creation routines, IDL creates a *widget record* that represents one or more native widgets in the current platform's user interface toolkit. The widget record is used to contain all of the information needed by IDL to manage the widget's state. The platform-specific native widget that underlies the IDL widget is not created (or *instantiated*) until you call the WIDGET_CONTROL procedure with the REALIZE keyword (see "Manipulating Widgets" on page 747 for details on using WIDGET_CONTROL). Between the time when the widget is created as an IDL widget record and when it is realized as a platform-specific interface element, you have control over many, but not all, aspects of the widget's state. Some details of the final realized widget's state (such as its exact screen geometry) may remain undetermined until the widget is instantiated.

It is important to note that unrealized widgets in a widget hierarchy can be manipulated programmatically. Examples of attributes you can manipulate before realization are the overall geometry of the user interface, widget values, and user values. You can even retrieve widget values before the widgets are realized. Unrealized widgets do not, however, generate widget events, since the actual platform-specific user interface has yet to be created.

Once a widget has been realized, its corresponding platform-specific user interface toolkit element is instantiated. The native toolkit determines the widget's exact screen geometry. If the widget is then *mapped*, it becomes visible on the computer screen, can be manipulated by a user, and generates widget events.

Note

Widgets are mapped by default. This means that when you realize a widget hierarchy, the widgets included in that hierarchy will usually be displayed on screen immediately. You can control the visibility of widget hierarchies — before or after

realization — using the MAP keyword to WIDGET_CONTROL. See “Controlling Widget Visibility” on page 748 for details.

Note also that widgets that are visible on screen can be made unavailable to the user by setting the SENSITIVE keyword to WIDGET_CONTROL. See “Sensitizing Widgets” on page 749 for details.

Example 1: A Simple Widget Application

The following example demonstrates the simplicity of widget programming. The example program creates a base widget containing a single button, labelled “Done.” When you position the mouse cursor over the button and click, the widget is destroyed.

Note

If you are new to IDL widget programming, don’t be dismayed if parts of this example are not immediately clear to you. As you read further through this chapter, the principles of the event-driven programming model and IDL’s specific implementation of that model will become clearer.

Note

This example is included in the file `widget1.pro` in the `examples/widgets` subdirectory of the IDL distribution. You can either open the file in an IDL editor window and compile and run the code using items on the **Run** menu, or simply enter

```
widget1
```

at the IDL command prompt. See “Running the Example Code” on page 738 if IDL does not run the program as expected.

```
PRO widget1_event, ev
  IF ev.SELECT THEN WIDGET_CONTROL, ev.TOP, /DESTROY
END

PRO widget1
  base = WIDGET_BASE(/COLUMN)
  button = WIDGET_BUTTON(base, value='Done')
  WIDGET_CONTROL, base, /REALIZE
  XMANAGER, 'widget1', base
END
```

While this simple example does nothing particularly useful, it does illustrate some basic concepts of event-driven programming. Let’s examine how the example is constructed.

First, note that the “application” consists of two parts: an event handling routine and a creation routine. Let’s first examine the second part — the creation routine — contained in the `widget1` procedure.

The `widget1` procedure does the following:

1. Creates a top-level base widget whose widget ID is stored in the variable `base`. All widget applications have at least one base.

2. Creates a button widget whose widget ID is stored in the variable `button`. The button widget has `base` as its parent. The value “Done” is assigned to the button. The value of a button widget is the text that appears on the button’s face.
3. Realizes the widget hierarchy built on `base` by calling WIDGET_CONTROL with the /REALIZE keyword. Realizing the widget hierarchy displays the widget on your computer screen.
4. Invokes the XMANAGER routine to manage the widget event loop, providing the name of the calling routine (`widget1`) and the widget ID of the top-level base on which the widget hierarchy is built (`base`).

The `widget1_event` procedure is the event handling routine for the application. By convention, the XMANAGER procedure looks for an event handling procedure with the same name as the procedure that creates the widgets, with “_event” appended to the end. (This default can be overridden by specifying an event handler directly using the EVENT_HANDLER keyword to XMANAGER.) When an event is received by XMANAGER, the event structure is passed to the `widget1_event` procedure via the `ev` argument.

In this example, all the event handling routine does is check the event structure to see if the event passed to it was a select event generated by the button widget. If a `SELECT` event is received, the routine calls WIDGET_CONTROL with the DESTROY keyword to destroy the widget hierarchy built on the top-level base widget (specified in the `TOP` field of the event structure).

For further discussion of widget events and event structures, see “Widget Event Processing” on page 755. For details about the event structures returned by different widgets, see the documentation for each widget in the *IDL Reference Guide*.

Widget Application Lifecycle

When you create and use a widget application, you do the following things:

1. Construct the Widget Hierarchy
2. Provide an Event-Handling Routine
3. Realize the Widgets
4. Register the Program with the XMANAGER
5. Interact with the Application
6. Destroy the Widgets

Construct the Widget Hierarchy

You must first build a widget hierarchy using the WIDGET_* functions. Start by creating a *top-level base* with the WIDGET_BASE function.

Combine other widget creation functions — WIDGET_BUTTON, CW_PDMENU, etc. — to create and organize the user interface of your widget application. At this point, the widgets are *unrealized* — they exist only as IDL widget records — and nothing has been created or displayed on the screen.

Note

Widget applications can include multiple widget hierarchies headed by multiple top-level base widgets. See “Using Multiple Widget Hierarchies” on page 787 for more on creating a hierarchy of widget hierarchies.

Provide an Event-Handling Routine

In order for a widget application to *do* anything, you must provide a routine that examines events, determines what action to take, and implements that action. Actions may involve computation, graphics display, or updates to the widget interface itself.

For best performance, event processing routines must run and return to the calling routine as quickly as possible. Widgets won’t respond to user input while the event-processing routine is running. Widget-based programs should wait for user-generated events, handle them as quickly as possible, and return to wait for more events. Event processing is discussed in detail in “Widget Event Processing” on page 755.

Event handling routines can manipulate widgets via the WIDGET_CONTROL procedure. Possible actions include the following:

- Obtain or change the value of a widget (see “Widget Values” on page 707) using the APPEND, GET_VALUE, and SET_VALUE keywords.
- Obtain or change the value of a widget’s user value using the GET_UVALUE and SET_UVALUE keywords. (User values are discussed in “Widget User Values” on page 754)
- Map and unmap widgets using the MAP keyword. Unmapped widgets are removed from the screen and become invisible, but they still exist in memory.
- Change a widget’s sensitivity using the SENSITIVE keyword. A widget indicates that it is insensitive by changing its appearance (often by graying itself or displaying text with dashed lines) and ignoring any user input. It is useful to make widgets insensitive at points where it would be inconvenient to get events from them (for example, if your program is waiting for input from another source).
- Change the settings of toggle buttons using the SET_BUTTON keyword.
- Push a widget hierarchy behind the other windows on the screen, or pull it in front, using the SHOW keyword.
- Display the “hourglass” cursor while the application is busy and not able to respond to user actions by setting the HOURGLASS keyword. (See “Indicating Time-Consuming Operations” on page 749.)

Realize the Widgets

To convert the IDL widget records representing your widget hierarchy into a set of platform-specific user interface toolkit elements, use the REALIZE keyword to the WIDGET_CONTROL procedure. Unless you have specifically *unmapped* the widgets before realizing them, the REALIZE keyword causes the widgets to be displayed on screen. See “Manipulating Widgets” on page 747 for additional details.

Register the Program with the XMANAGER

Your widget application waits for events to be reported to it and reacts as specified in the event handling routine after being registered with the XMANAGER procedure.

Events are obtained by XMANAGER via the WIDGET_EVENT function and passed to the calling routine (your event handler) in the form of an IDL structure variable. Each type of widget returns a different type of structure, as described in the documentation for the individual widget creation functions in the *IDL Reference Guide*. Every event structure has three common elements: long integers named `ID`, `TOP`, and `HANDLER`:

- `ID` is the widget ID of the widget generating the event.
- `TOP` is the widget ID of the top-level base containing the widget that generated the event.
- `HANDLER` is important for event handler functions, which are discussed later in this chapter.

When an event occurs, XMANAGER arranges for the event structure to be passed to an event-handling procedure specified by the program, and the event handler takes some appropriate action based on the event. This means that multiple widget applications can run simultaneously — XMANAGER arranges for the events be dispatched to the appropriate routine.

Interact with the Application

Once the widget application has been realized and registered with XMANAGER, the user can interact with the application to accomplish whatever tasks the application is designed to accomplish.

Destroy the Widgets

When the application has finished (usually when the user clicks on a “Done” or “Quit” button), destroy the widget hierarchy using the DESTROY keyword to the WIDGET_CONTROL procedure. This causes all resources related to the hierarchy to be freed and removes it from the screen.

Manipulating Widgets

IDL provides several routines that allow you to manipulate and manage widgets programmatically:

- WIDGET_CONTROL allows you to realize widget hierarchies, manipulate them, and destroy them.
- WIDGET_EVENT allows you to process events generated by a specific widget hierarchy.
- WIDGET_INFO allows you to obtain information about the state of a specific widget or widget hierarchy.
- XMANAGER provides an event loop and manages events generated by all existing widget hierarchies.
- XREGISTERED allows you to test whether a specific widget is currently registered with XMANAGER.

These widget manipulation routines are discussed in more detail in the following sections.

WIDGET_CONTROL

The WIDGET_CONTROL procedure allows you to realize, manage, and destroy widget hierarchies. It is often used to change the default behavior or appearance of previously-realized widgets.

Keywords to WIDGET_CONTROL may affect only certain types of widgets, any type of widget, or the widget system in general. See "WIDGET_CONTROL" in the *IDL Reference Guide* manual for complete details. We discuss here only a few of the more common uses of this procedure.

Realizing Widget Hierarchies

IDL widgets are actually *widget records* that represent platform-specific user interface toolkit elements. In order to instantiate the platform-specific toolkit elements, widgets must be *realized* with the following statement:

```
WIDGET_CONTROL, base, /REALIZE
```

where `base` is the widget ID of the top-level base widget for your widget hierarchy.

Destroying Widget Hierarchies

The standard way to destroy a widget hierarchy is with the statement:

```
WIDGET_CONTROL, base, /DESTROY
```

where `base` is the widget ID of the top-level base widget of the hierarchy to be killed. Usually, IDL programs that use widgets issue this statement in their event-handling routine in response to the user's clicking on a "Done" button in the application.

In addition, some window managers place a pulldown menu on the frame of the top-level base widget that allows the user to kill the entire hierarchy. Using the window manager to kill a widget hierarchy is equivalent to using the DESTROY keyword to the WIDGET_CONTROL procedure.

When designing widget applications, you should always include a "Done" button (or some other widget that allows the user to exit) in the application itself, since some window managers do not provide the user with a kill option from the outer frame.

Retrieving or Changing Widget Values

You can use WIDGET_CONTROL to retrieve or change widget values using the GET_VALUE and SET_VALUE keywords. Similarly, you can retrieve or change widget user values with the GET_UVALUE and SET_UVALUE keywords.

For example, you could use the following commands to retrieve the value of a draw widget whose widget ID is stored in the variable `drawwid`, and to make that draw widget the current graphics window:

```
WIDGET_CONTROL, drawwid, GET_VALUE=draw
WSET, draw
```

Similarly, you could use the following command in an event handling procedure to save the user value of the widget that generates an event into an IDL variable named `uval`:

```
WIDGET_CONTROL, event.id, GET_UVALUE=uval
```

For more on widget user values, see "Widget User Values" on page 754.

Controlling Widget Visibility

You can display or remove realized widgets from the screen by *mapping* or *unmapping* them. Unmapped widgets still exist in the widget hierarchy, but they are not displayed and do not generate events.

Set the MAP keyword to WIDGET_CONTROL equal to zero to hide a widget, or to a nonzero value to display it again. For example, to hide the `base1` widget and all its child widgets from view, use the following command:

```
WIDGET_CONTROL, base1, MAP=0
```

By default, widgets are mapped automatically when they are realized. You can prevent a widget from appearing on screen when you realize it by setting MAP=0 *before* realizing the widget hierarchy.

Note

While it is possible to call WIDGET_CONTROL, MAP=0 with the widget ID of any widget, only base widgets can actually be unmapped. If you specify a widget ID that is not from a base widget, IDL searches upward in the widget hierarchy until it finds the closest base widget. The map operation is applied to that base.

Sensitizing Widgets

Use sensitivity to control when a user is allowed to manipulate a widget. When a widget is sensitive, it has a normal appearance and can receive user input. When a widget is insensitive, it ignores any input directed at it. Note that while most widgets change their appearance when they become insensitive, some simply stop generating events.

Set the SENSITIVE keyword equal to zero to desensitize a widget, or to a nonzero value to make it sensitive. For example, you might wish to make a group of buttons contained in a base whose widget ID is stored in the variable `bgroup` insensitive after some user input. You would use the following command:

```
WIDGET_CONTROL, bgroup, SENSITIVE=0
```

Indicating Time-Consuming Operations

In an event driven environment, it is important that the interface be highly responsive to the user's manipulations. Widget event handlers should be written to execute quickly and return. However, sometimes the event handler has no option but to perform an operation that is slow. In such a case, it is a good idea to give the user feedback that the system is busy. This is easily done using the HOURGLASS keyword just before the expensive operation is started:

```
WIDGET_CONTROL, /HOURGLASS
```

This command causes IDL to turn on an hourglass-shaped cursor for all IDL widgets and graphics windows. The hourglass remains active until the next event is processed, at which point the previous cursor is automatically restored.

WIDGET_EVENT

The WIDGET_EVENT function returns events for the widget hierarchy rooted at *Widget_ID*. Events are generated when a button is pressed, a slider position is

changed, and so forth. In most cases, you will not use WIDGET_EVENT directly, but instead will use the XMANAGER routine to manage widget events. Event processing is discussed in detail in "Widget Event Processing" on page 755. See also "WIDGET_EVENT" in the *IDL Reference Guide* manual for additional details.

WIDGET_INFO

The WIDGET_INFO function is used to obtain information about the widget subsystem and individual widgets. You supply the widget ID of a widget for which you want to retrieve some information, along with a keyword that specifies the type of information. For example, to determine the index of the selected item in a list widget whose widget ID is contained in the variable `list`, you would use a command like the following:

```
listindex = WIDGET_INFO(list, /LIST_SELECT)
```

Finding Widget IDs using WIDGET_INFO

One noteworthy use of WIDGET_INFO is to locate the widget ID of a widget with a specified *user name*. (A *user name* is a part of the widget's widget record that contains a text identifier, specified by the programmer.) See "Working With Widget IDs" on page 752 for more information on this technique.

See "WIDGET_INFO" in the *IDL Reference Guide* manual for more information.

XMANAGER

The XMANAGER procedure provides the main event loop registration and widget management. Calling XMANAGER "registers" a widget program with the XMANAGER event handler. XMANAGER takes control of event processing until all widgets have been destroyed.

Using XMANAGER allows you to run multiple widget applications and work at the IDL command line at the same time. While it is possible to use WIDGET_EVENT directly to manage events in your application, it is almost always easier to use XMANAGER.

See "XMANAGER" in the *IDL Reference Guide* manual for complete details.

XREGISTERED

The XREGISTERED function returns True if the widget specified by its argument is currently registered with the XMANAGER.

One use of the XREGISTERED function is to control the number of instances of a given widget application that run at a given time. For example, suppose that you have a widget program that registers itself with the XMANAGER with the command:

```
XMANAGER, 'mywidget', base
```

You could limit this widget to one instantiation by adding the following line as the first line (after the procedure definition statement) of the widget creation routine:

```
IF (XREGISTERED('mywidget') NE 0) THEN RETURN
```

See "XREGISTERED" in the *IDL Reference Guide* manual for complete details.

Working With Widget IDs

Any widget application capable of doing real work will include one or more routines that are separate from the routine that creates the widget hierarchy, designed to handle and respond to user-generated events. *Event processing routines* — the routines that process information contained in widget event structures and respond accordingly — often retrieve information contained in the widget values of the widgets that make up the interface, perform calculations, and modify the widget interface itself in response to user actions.

Since a widget ID is required to retrieve information from or set values in a widget, you will need a way for your event processing routines to retrieve the ID of a specified widget. This section describes techniques you can use to pass widget IDs between the routines in your widget application — most notably between the widget creation routine (where widget IDs are generated) and the event processing routines.

Use the Widget Event Structure

Every time a user interacts with a widget using the mouse or keyboard, a *widget event structure* is generated. Widget event structures contain the widget ID of the widget that generated the event. In addition, widget event structures provide the widget ID of the top-level base in the widget hierarchy to which the widget the generated the event belongs.

Getting the widget ID of the appropriate widget from the event structure is almost always the preferred method for passing a widget ID from one routine to another within your application. Widget event processing is discussed in detail in “Widget Event Processing” on page 755.

Pass the Widget ID Using a Widget User Value

The widget event structure always includes two widget IDs: the ID of the widget that generated the event, and the ID of the top-level base widget. If you need to pass multiple widget IDs between routines, it is often useful to place the widget ID values in the *user value* of the top-level base widget. Widget user values are discussed in “Widget User Values” on page 754.

Use a User Name to Locate the Widget

One of the pieces of information you can specify when you create a widget is a *user name*. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID. To specify

a user name, set the UNAME keyword to the widget creation routine equal to a string that can be used to identify the widget in your code.

To query the widget hierarchy, use the WIDGET_INFO function with the widget ID of the top-level base widget and the FIND_BY_UNAME keyword. Note that user names must be unique within the widget hierarchy, because the FIND_BY_UNAME keyword returns the ID of the first widget with the specified name.

Pass the Widget ID Explicitly

In some cases, you may need to pass a specific widget ID available in one routine to a second routine. In this case, you can specify the widget ID as a parameter when calling the second routine from the first. While this method is not so general as using the widget event structure, it is useful in some circumstances.

Use a COMMON Block

In rare cases, it may be useful to store widget IDs in a COMMON block, making them available to all routines in the application. While using a COMMON block may seem like a good strategy on first inspection, this method has several drawbacks. Most importantly, using a COMMON block to hold widget IDs means that only one instance of a given widget application can be running at once.

Widget User Values

Every widget primitive and compound widget can carry a user-specified value of any IDL data type and organization; that is, every widget contains a variable that can store arbitrary information. This value is ignored by the widget and is for the programmer's convenience only.

The initial *user value* is specified using the UVALUE keyword to the widget creation function. If no initial value is specified, the user value is undefined. Once the widget exists, its user value can be examined and/or changed using the GET_UVALUE and SET_UVALUE keywords to the WIDGET_CONTROL procedure.

Note

The *widget user value* should not be confused with the *widget value*, described in "Widget Values" on page 707.

User Values Simplify Event Handling

User values can be used to simplify event-handling. If each widget has a distinct user value, you need only check the user value of any event to determine which widget generated it. In practice, this means you do not need to keep track of the widget IDs of all the widgets in your widget hierarchy in order to determine what to do with a given event.

User Values can be Accessible Throughout a Widget Application

Another use for user variables is to simulate a variable that is available in more than one IDL routine. For example, you can set the user value of a top-level base widget equal to one or more widget IDs. You then have an easy way to pass the widget IDs from your widget creation routine to your event handling routine.

We will take advantage of both of these aspects of user values in "Example 2: Event Processing and User Values" on page 761.

Widget Event Processing

The concepts of events and event processing underlie every aspect of widget programming. It is important to understand how IDL handles widget events in order to use widgets effectively.

This section discusses the following topics:

- What are Widget Events?
- Structure of Widget Events
- Managing Widget Events with XMANAGER
- Event Processing and Callbacks

For a discussion of techniques you can use to detect and respond to specific types of events, see "Working with Widget Events" in Chapter 27.

What are Widget Events?

A widget event is a message returned from the window system when a user manipulates a widget. In response to an event, a widget program usually performs some action (*e.g.*, opens a file, updates a plot).

Structure of Widget Events

As events arrive from the window system, IDL saves them in a queue for the target widget. The WIDGET_EVENT function delivers these events to the IDL program as IDL structures. Every widget event structure has the same first three fields: these are long integers named `ID`, `TOP`, and `HANDLER`:

- `ID` is the widget ID of the widget that generated the event.
- `TOP` is the widget ID of the top-level base containing `ID`.
- `HANDLER` is the widget ID of the widget associated with the event handling routine. The importance of `HANDLER` will become apparent when we discuss event routines and compound widgets, below.

Event structures for different widgets may contain other fields as well. The exact form of the event structure for any given widget is described in the documentation for that widget's creation function in the *IDL Reference Guide*.

Managing Widget Events with XMANAGER

The XMANAGER procedure provides a convenient, simplified interface IDL's event-handling capabilities. At the highest level, creating a widget application consists of the following steps:

1. Creating routines to react to widget events.
2. Creating the widgets that make up the application's interface.
3. Realizing the widgets.
4. Calling XMANAGER to manage events flowing from the widget interface.

XMANAGER arranges for an event-handling procedure supplied by the application to be called when events for it arrive. The application is shielded from the details of calling the underlying WIDGET_EVENT function and interacting with other widget applications that may be running simultaneously.

Note

While it is possible for a user-written program to call the WIDGET_EVENT function directly, in practice this is very unusual. For details on how events are handled at a low level, see "The WIDGET_EVENT Function" on page 758.

The file `xmng_tmpl.pro`, found in the `lib` subdirectory of the IDL distribution, is a template for writing widget applications that use XMANAGER.

XMANAGER and *Blocking*

The term *blocking* is used to describe a situation in which processing by IDL is suspended until some event or action takes place. Unless you specifically arrange otherwise, IDL will only allow one user interface (the IDL command line or a single widget application) to be active at one time. XMANAGER simplifies the process of arranging things so that multiple user interfaces can run at the same time — that is, managing events so that applications do not need to *block* in order to be assured of receiving the correct event information.

IDL's blocking behavior is discussed in detail in "XMANAGER" in the *IDL Reference Guide* manual. In most cases, specifying the NO_BLOCK keyword when calling XMANAGER will allow your application to "play nicely with others," but you should keep the following things in mind when writing widget applications:

Active Command Line

IDL can provide an *active command line*. If the command line is active, IDL will execute commands entered at the command line even if one or more widget

applications are already running. In order for IDL to behave in this way, all widget applications must be run via XMANAGER with the NO_BLOCK keyword set. See "Active Command Line" under "XMANAGER" in the *IDL Reference Guide* manual for details

Blocking and Non-Blocking Applications

By default, widget applications — even those managed with XMANAGER — will block. To enable your application to run without blocking other widget applications or the IDL command line, you must explicitly set the NO_BLOCK keyword to XMANAGER when registering the application. Put another way, any running widget application that does not have this keyword set will block all event processing for widget applications and the IDL command line. See "Blocking vs. Non-blocking Applications" under "XMANAGER" in the *IDL Reference Guide* manual for details.

Registering Applications Without Processing Their Events

In order to allow multiple widget applications to run simultaneously, each application must be *registered* with XMANAGER, so it knows how to recognize events generated by the application. In most cases, the registration step takes place automatically when XMANAGER is called to begin processing events for the application.

In some cases, however, it may be useful to register an application with XMANAGER before asking it to begin processing the application's events. In these cases, you can use the JUST_REG keyword to XMANAGER; the application is added to XMANAGER's list of known applications without starting event processing, and XMANAGER returns immediately. See "JUST_REG vs. NO_BLOCK" under "XMANAGER" in the *IDL Reference Guide* manual for details.

Tips on Working With XMANAGER

Because XMANAGER buffers you from direct handling of widget events, you *cannot* explicitly specify an event-handling function or procedure for the top-level base using the EVENT_FUNC or EVENT_PRO keywords to WIDGET_BASE or WIDGET_CONTROL. Event handlers for top-level bases specified via these keywords will be overwritten by XMANAGER.

Instead, provide the name of the event handler routine to XMANAGER via the EVENT_HANDLER keyword. If you do not supply the name of an event handler via the EVENT_HANDLER keyword, XMANAGER will construct a default name by adding the suffix "_event" to the *Name* argument.

Note that this guideline applies only to top-level bases (base widgets created with no parent widget). Child base widgets should use the EVENT_FUNC or EVENT_PRO keywords to specify event handling routines, if necessary.

In addition, it is often convenient to specify the death-notification routine for the top-level base of a widget application via the CLEANUP routine to XMANAGER rather than via the KILL_NOTIFY keyword to WIDGET_BASE or WIDGET_CONTROL. Either method will work, but the *last* routine specified is the routine that will be called when the base widget is destroyed. Since the call to XMANAGER is often the last call made when creating a widget application, using the CLEANUP keyword to specify the routine to be called when the application is ends is preferred.

The XREGISTERED Function

The XMANAGER procedure allows multiple instances of a widget application to run simultaneously. In some cases, however, you may wish to ensure that only a single instance of application can run at a given time. An obvious example of this is an application that uses a COMMON block to maintain its current state (see "Managing Application State" on page 763).

The XREGISTERED function can be used in such applications to ensure that only a single copy of the application run at a time. Place the following statement at the start of the widget creation routine:

```
IF (XREGISTERED('routine_name') NE 0) THEN RETURN
```

where *routine_name* is the name of the widget application.

See "XREGISTERED" in the *IDL Reference Guide* manual for further information.

The WIDGET_EVENT Function

All widget event processing in IDL is eventually handled by the WIDGET_EVENT function. Note that while we will discuss WIDGET_EVENT here for completeness, in most cases you will *not* want to call WIDGET_EVENT directly. The XMANAGER routine provides a convenient, simplified interface to WIDGET_EVENT and allows IDL to take over the task of managing multiple widget applications.

In its simplest form, the WIDGET_EVENT function is called with a widget ID (usually, the ID of a base widget) as its argument. WIDGET_EVENT checks the queue of undelivered events for that widget *or any of its children*. If an event is present, it is immediately dequeued and returned. If no event is available, WIDGET_EVENT blocks all other processing by IDL until an event arrives, and then returns it. Typically, the request is made for a top-level base, so WIDGET_EVENT returns events for any widget in the widget hierarchy rooted at that base widget.

This simple usage suffers from a major weakness. Since each call to WIDGET_EVENT is looking for events from a specified widget hierarchy, it is not possible to receive events for more than one widget hierarchy at a time. It is important

to be able to run multiple widget applications (each with a separate top-level base) simultaneously. An example would be an image processing application, a colortable manipulation tool, and an on-line help reader all running together.

One solution to this problem is to call WIDGET_EVENT with an array of widget identifiers instead of a single ID. In this case, WIDGET_EVENT returns events for any widget hierarchy in the list. This solution is effective, but it still requires that you maintain a complete list of all interesting top-level base identifiers, which implies that all cooperating applications need to know about each other.

The most powerful way to use WIDGET_EVENT is to call it without any arguments at all. Called this way, it will return events for any currently-realized widgets that have expressed an interest in being managed. (You specify that a widget wants to be managed by setting the MANAGED keyword to the WIDGET_CONTROL procedure.) This form of WIDGET_EVENT is especially useful when used in conjuction with widget event callback routines, discussed in "Event Processing and Callbacks" on page 759.

Event Processing and Callbacks

Previously, we mentioned that when IDL receives an event, the event is queued until a call to WIDGET_EVENT is made (either explicitly by the user program or by XMANAGER), whereupon the event is dequeued and returned. The following is a more complete description of what actually happens in IDL's *event loop*.

Events for a given widget are processed in the order that they are generated. The event processing performed by WIDGET_EVENT consists of the following steps, applied iteratively:

1. Wait for an event from one of the specified widgets to arrive.
2. Starting with the widget that generated the event, search up the widget hierarchy for a widget with an associated event-handling procedure or function.

   Event-handling routines associated with widgets are known as *callback* routines. Other cases where an IDL system routine (WIDGET_EVENT, in this instance) calls a user-specified, user-written routine include routines specified via the KILL_NOTIFY or NOTIFY_REALIZE keywords to the widget creation functions and WIDGET_CONTROL, as well as the corollary keywords to XMANAGER.
3. If an event-handling *procedure* is found, it is called with the event as its argument. The `HANDLER` field of the event is set to the widget ID of the widget associated with the handling procedure. When the procedure returns,

WIDGET_EVENT returns to the first step above and starts searching for events. Hence, event-handling procedures are said to “swallow” events.

4. If an event-handling *function* is found, it is called with the event as its argument. The `HANDLER` field of the event is set to the widget ID of the widget associated with the handling function.

   When the function returns, its value is examined. If the value is not a structure, it is discarded and WIDGET_EVENT returns to the first step. This behavior allows event-handling functions to selectively act like event-handling procedures and “swallow” events.

   If the returned value is a structure, it is checked to ensure that it has the standard first three fields: `ID`, `TOP`, and `HANDLER`. If any of these fields is missing, IDL issues an error. Otherwise, the returned value replaces the event found in the first step and WIDGET_EVENT continues moving up the widget hierarchy looking for another event handler routine, as described in step 2, above.

   In situations where an event structure is returned, event functions are said to “rewrite” events. This ability to rewrite events is the basis of *compound widgets*, which combine several widgets to give the appearance of a single, more complicated widget. Compound widgets are an important widget programming concept. For more information, see “Compound Widgets” on page 767.

5. If an event reaches the top of a widget hierarchy without being swallowed by an event handler, it is returned as the value of WIDGET_EVENT.

6. If WIDGET_EVENT was called without an argument, and there are no widgets left on the screen that are being managed (as specified via the MANAGED keyword to the WIDGET_CONTROL procedure) and could generate events, WIDGET_EVENT ends the search and returns an *empty event* (a standard widget event structure with the top three fields set to zero).

Example 2: Event Processing and User Values

The following example demonstrates how user values can be used to simplify event processing and to pass values between routines. It creates a base widget with three buttons and a text field that reports which button was pressed.

Note

If you are new to IDL widget programming, don't be worried if parts of this example are not immediately clear to you. As you read further through this chapter, the principles of the event-driven programming model and IDL's specific implementation of that model will become clearer.

Note

This example is included in the file `widget2.pro` in the `examples/widgets` subdirectory of the IDL distribution. You can either open the file in an IDL editor window and compile and run the code using items on the **Run** menu, or simply enter

```
widget2
```

at the IDL command prompt. See "Running the Example Code" on page 738 if IDL does not run the program as expected.

```
PRO widget2_event, ev
  WIDGET_CONTROL, ev.TOP, GET_UVALUE=textwid
  WIDGET_CONTROL, ev.ID, GET_UVALUE=uval
  CASE uval OF
    'ONE' : WIDGET_CONTROL, textwid, SET_VALUE='Button 1 Pressed'
    'TWO' : WIDGET_CONTROL, textwid, SET_VALUE='Button 2 Pressed'
    'DONE': WIDGET_CONTROL, ev.TOP, /DESTROY
  ENDCASE
END

PRO widget2
  base = WIDGET_BASE(/COLUMN)
  button1 = WIDGET_BUTTON(base, VALUE='One', UVALUE='ONE')
  button2 = WIDGET_BUTTON(base, VALUE='Two', UVALUE='TWO')
  text = WIDGET_TEXT(base, XSIZE=20)
  button3 = WIDGET_BUTTON(base, value='Done', UVALUE='DONE')
  WIDGET_CONTROL, base, SET_UVALUE=text
  WIDGET_CONTROL, base, /REALIZE
  XMANAGER, 'widget2', base
END
```

Let's examine the creation routine, `widget2`, first. We first create a top-level base, this time specifying the COLUMN keyword to ensure that the widgets contained in the base are stacked vertically. We create two buttons with values "One" and "Two,"

and user values “ONE” and “TWO.” Remember that the *value* of a button widget is also the button’s label. We create a text widget, and specify its width to be 20 characters using the XSIZE keyword. The last button is the “Done” button, with a the user value “DONE.”

Next follow two calls to the WIDGET_CONTROL procedure. The first call sets the user value of the top-level base widget equal to the widget ID of our text widget, allowing easy access to the text widget from the event handling routine. The second call realizes the top-level base and all its child widgets. Finally, we invoke the XMANAGER to manage the widget application.

The `widget2_event` routine is slightly more complicated than the event handler in “Example 1: A Simple Widget Application” on page 742, but it is still relatively simple. We begin by using WIDGET_CONTROL to retrieve the widget ID of our text widget from the user value of the top-level base. We can do this because the widget ID of our top-level base is contained in the `TOP` field of the widget event structure. We use the GET_UVALUE keyword to store the widget ID of the text widget in the variable `textwid`.

Next, we use WIDGET_CONTROL with the GET_UVALUE keyword to retrieve the user value of the widget that generated the event. Again, we can do this because we know that the widget ID of the widget that generated the event is stored in the `ID` field of the event structure. We then use a CASE statement to compare the user value of the widget, now stored in the variable `uval`, with the list of possible user values to determine which button was pressed and act accordingly.

In the CASE statement, we check to see if `uval` is the user value associated with either button one or button two. If it is, we use WIDGET_CONTROL and the SET_VALUE keyword to alter the value of the text widget, whose ID we stored in the variable `textwid`. If `uval` is `'DONE'`, we recognize that the user has clicked on the “Done” button and use WIDGET_CONTROL to destroy the widget hierarchy.

Managing Application State

A widget application is usually divided into at least two separate routines, one that creates and realizes the application and another that handles events. These multiple routines need shared access to certain types of information, such as the widget IDs of the application's widgets and data being used by the application. This shared information is referred to as the *application state*.

Techniques for Preserving Application State

The following are some techniques you can use to preserve and share application state data between routines.

Using COMMON Blocks

One obvious answer to this problem is to use a COMMON block to hold the state. However, this solution is undesirable because it prevents more than a single copy of the application from running at the same time. It is easy to imagine the chaos that would ensue if multiple instances of the same application were using the *same* common block without some sort of interlocking.

Using a State Structure in a User Value

A better solution to this problem is to use the user value of one of the widgets to store state information for the application.Using this technique, multiple instances of the same widget code can exist simultaneously. Since this user value can be of any type, a structure can be used to store any number of state-related values.

For example, consider the following example widget code:

```
PRO my_widget_event, event
  WIDGET_CONTROL, event.TOP, GET_UVALUE=state, /NO_COPY

  Event-handling code goes here

  WIDGET_CONTROL, event.TOP, SET_UVALUE=state, /NO_COPY
END

PRO my_widget
  ; Create some widgets
  wBase = WIDGET_BASE(/COLUMN)
  wDraw = WIDGET_DRAW(wBAse, XSIZE=300, YSIZE=300)

  ; Realize the base widget and retrieve the widget ID
  ; of the drawable area.
```

```
  WIDGET_CONTROL, wBase, /REALIZE
  WIDGET_CONTROL, wDraw, GET_VALUE=idxDraw

  ; Create a state structure variable and set the user
  ; value of the top-level base equal to the state variable.
  state = {wDraw:wDraw, idxDraw:idxDraw}
  WIDGET_CONTROL, wBase, SET_UVALUE=state

  ; Use XMANAGER to manage the widgets
  XMANANAGER, 'my_widget', wBase
END
```

In this example, we store state information (the widget ID of the draw widget and the index of the drawable area) in a structure variable, and set the user value of the top-level base widget equal to that structure variable. This makes it possible to retrieve the structure using the widget ID contained in the `TOP` field of any widget event structure that arrives at the event handler routine.

Notice the use of the NO_COPY keyword to WIDGET_CONTROL in the example. This keyword prevents IDL from duplicating the memory used by the user value during the GET_UVALUE and SET_UVALUE operations. This is an important efficiency consideration if the size of the state data is large. (In this example the use of NO_COPY is not really necessary, as the state data consists only of the two long integers that represent the widget IDs being passed in the state variable.)

While it is important to consider efficiency, the use of the NO_COPY keyword does have the side effect of causing the user value of the widget to become undefined when it is retrieved using the GET_UVALUE keyword. If the user value is not replaced before the event handler exits, the next execution of the event routine will fail, since the user value will be undefined.

Using a Pointer to the State Structure

A variation on the above technique uses an IDL pointer to contain the state variable. This eliminates the duplication of data and the need for the use of the NO_COPY keyword.

Consider the following example widget code:

```
PRO my_widget_event, event
  WIDGET_CONTROL, event.TOP, GET_UVALUE=pState

  Event-handling code goes here, accessing the state
  structure via the retrieved pointer.

END

PRO my_widget_cleanup, wBase
```

```
  ; This routine is called when the application quits.
  ; Retrieve the state variable and free the pointer.
  WIDGET_CONTROL, wBase, GET_UVALUE=pState
  PTR_FREE, pState
END

PRO my_widget
  ; Create some widgets.
  wBase = WIDGET_BASE(/COLUMN)
  wDraw = WIDGET_DRAW(wBAse, XSIZE=300, YSIZE=300)

  ; Realize the base widget and retrieve the widget ID
  ; of the drawable area.
  WIDGET_CONTROL, wBase, /REALIZE
  WIDGET_CONTROL, wDraw, GET_VALUE=idxDraw

  ; Create a state structure variable.
  state = {wDraw:wDraw, idxDraw:idxDraw}

  ; Place the state structure in a pointer and set the user
  ; value of the top-level base widget equal to the pointer.
  pState = PTR_NEW(state, /NO_COPY)
  WIDGET_CONTROL, wBase, SET_UVALUE=pState, /NO_COPY

  ; Call XMANAGER to manage the widgets, specifying the routine
  ; to be called when the application quits.
  XMANANAGER, 'my_widget', wBase, CLEANUP='my_widget_cleanup'
END
```

Notice the following differences between this technique and the technique shown in the previous example:

- This method eliminates the removal of the user value from the top-level base widget by removing the use of the NO_COPY keyword with the GET_UVALUE keyword to WIDGET_CONTROL. Since only the pointer (a long integer) is passed to the event routine, the efficiency issues connected with copying the value are small enough to ignore. (Note that we do use the NO_COPY keyword when creating the pointer and when initially setting the user value of the top-level base widget; since these statements are executed only once, we don't worry about the fact that the `state` or `pState` variables become undefined.)
- The state structure contained in the pointer must now be referenced using pointer-dereferencing syntax. For example, to refer to the `idxDraw` field of the state structure within the event-handling routine, you would use the syntax

  ```
  (*pState).idxDraw
  ```

- The pointer allocated to store the state structure must be freed when the widget application quits. We do this by specifying a cleanup routine via the CLEANUP keyword to XMANAGER. It is the cleanup routine's responsibility to free the pointer.

Each of the above techniques has advantages. Choose a method based on the complexity of your application and your level of comfort with features like IDL pointers and the NO_COPY keyword.

Compound Widgets

Widget primitives can be used to construct many varied user interfaces, but complex programs written with them suffer the following drawbacks:

- Large widget applications become difficult to maintain. As an application grows, it becomes more difficult to properly write and test. The resulting program suffers from poor organization.
- Good ideas can be difficult to reuse. Most larger applications are constructed from smaller sub-units. For example, a color table editor might contain control panel, color selection and color-index selection sub-units. These sub-units are often complicated tools that could be used profitably in other programs. To reuse such sub-units, the programmer must understand the existing application and then transplant the interesting parts into the new program — at best a tedious and error-prone proposition.

Compound widgets solve these problems. A compound widget is a complete, self-contained, reusable widget sub-tree that behaves to a large degree just like a primitive widget. Complex widget applications written with compound widgets are much easier to maintain than the same application written without them. Using compound widgets is analogous to using subroutines and functions in programming languages.

Writing Compound Widgets

Compound widgets are written in the same way as any other widget application. They are distinguished from regular widget applications in the following ways:

- Compound widgets usually have a base widget at the root of their hierarchies. This base contains the subwidgets that make up the compound widget. From the user's point of view, this single widget *is* the compound widget — its children are not programmatically accessible on their own.

  Notice that the base widget at the root of a compound widget is *not* a top-level base. When used, a compound widget must always have a parent widget.
- It is important that the compound widget not make use of the base's user value. In order to preserve the illusion that the compound widget works just like any of the widget primitives, the user value of the compound widget's top-level base should be reserved for use by the caller of the compound widget. Instead, the compound widget should use the user value of one of its child widgets.
- The widget at the root of the compound widget's hierarchy *always* has an event handler function associated with it via the EVENT_FUNC keyword to the

widget creating function or the WIDGET_CONTROL procedure. This event handler manages events from its sub-widgets and generates events for the compound widget. By swallowing events from the widgets that comprise the compound widget and generating events that represent the compound widget, it presents the illusion that the compound widget is acting like a widget primitive.

- If the compound widget has a value that can be set, it should be assigned a value setting procedure via the PRO_SET_VALUE keyword to the widget creating function or the WIDGET_CONTROL procedure.
- If the compound widget has a value that can be retrieved, it should be assigned a value retrieving function via the FUNC_GET_VALUE keyword to the widget creating function or the WIDGET_CONTROL procedure.

For an example of how a compound widget might be written, see "Example 3: Compound Widget" on page 770.

The HANDLER Field of the Widget Event Structure

Recall that when WIDGET_EVENT finds an event to return, it moves up the widget hierarchy looking for an event-handling routine registered to the widgets in between its current position and the top-level base of the widget application. If such a routine is found, it is called with the event as its argument, and the `HANDLER` field of this event is set to the widget ID of the widget where the event routine was found. Since compound widgets have event handlers associated with their root widget, the `HANDLER` field gives the event handler the widget ID of the root widget. This allows the event handler for a compound widget instance to easily locate the location of its state information relative to this root.

Storing State Information

IDL programmers are often tempted to store the state information directly in the user value of the root widget, but this is not a good idea. The user value of a compound widget is reserved for the user of the widget, just like any basic widget. Therefore, you should store the state information in the user value of one of the child widgets below the root. As a convention, the user value of the first child is often used, leading to event handlers structured as follows:

```
FUNCTION EVENT_FUNC, event
  ; Get state from the first child of the compound widget root:
  child = WIDGET_INFO(event.HANDLER, /CHILD)
  WIDGET_CONTROL, child, GET_UVALUE=state, /NO_COPY

  ; Execute event-handling code here.
```

```
    ; Restore the state information before exiting routine:
    WIDGET_CONTROL, child, SET_UVALUE=state, /NO_COPY

    ; Return result of function
    RETURN, result
END
```

Sometimes, an application will find that it needs to use the user value of all its child widgets for some other purpose, and there is no convenient place to keep the state information. One way to work around this problem is to interpose an extra base between the root base and the rest of the widgets:

```
ROOT = WIDGET_BASE(parent)
EXTRA = WIDGET_BASE(root)
```

In such an approach, the remaining widgets would all be children of EXTRA rather than ROOT.

Example 3: Compound Widget

The following example incorporates ideas from the previous sections to show how you might approach the task of writing a compound widget. The widget is called CW_DICE, and it simulates a single six-sided die. Figure 26-1 shows the appearance of XDICE, an application that uses two instances of CW_DICE. XDICE is discussed in "Using CW_DICE in a Widget Program" on page 776.

Note

`cw_dice.pro` can be found in the lib subdirectory of the IDL distribution. `xdice.pro` can be found in the examples/widgets subdirectory of the IDL distribution. You should examine these files for additional details and comments not included here. We present sections of the code here for didactic purposes—there is no need to re-create either of these files yourself.

The CW_DICE compound widget has the following features:

- It uses a button widget. The current value of the die is displayed as a bitmap label on the button itself. When the user presses the button, the die "rolls" itself by displaying a sequence of bitmaps and then settles on a final value. An event is generated that returns this final value.
- Timer events are used to create the rolling effect. This allows the dice to give the same appearance on machines of varying performance levels. (Timer events are discussed in "Working with Widget Events" in Chapter 27.)
- The die can be set to a specific value via the SET_VALUE keyword to the WIDGET_CONTROL procedure. If the desired value is outside of the range 1 through 6, the die is rolled as if the user had pressed the button and a final value is selected randomly. Using WIDGET_CONTROL to set the value of the widget in this manner does not cause an event to be issued — IDL's convention is that user actions cause events, while programmatic changes do not.
- The current value of the die can be obtained via the GET_VALUE keyword to the WIDGET_CONTROL procedure.

Almost any compound widget will have an associated state. The following is the state of CW_DICE:

1. The current value.
2. The number of times the die should "tumble" before settling on a final value.
3. The amount of time to take between tumbles.

4. A count of how many tumbles are left before a final value is displayed, while a roll is in progress.
5. The bitmaps to use for the 6 possible die values.
6. The seed to use for the random number generator.

The first four items are stored in a per-widget structure kept in one of the child widget's user values. Since the bitmaps never change, it makes sense to keep them in a COMMON block to be accessed freely by all the CW_DICE routines. It also makes sense to use a single random number seed for the entire CW_DICE class rather than one per instance to avoid the situation where multiple dice, having been created at the same time, have the same seed and thus display the same value on each roll.

Note

It is rare that the use of a COMMON block in a compound widget makes sense. Notice, however, that we're only keeping read-only data (bitmaps) or data that can be overwritten at any time with no negative effects (random number generator seed).

Given the above decisions, it is now possible to write the CW_DICE procedure:

```
;Value is an optional argument that lets the caller set the initial
;die value to a value between 1 and 6. UVALUE will simply be passed
;on to the root base of CW_DICE. The TUMBLE keywords let the user
;adjust the tumble count and period.
FUNCTION cw_dice, parent, value, UVALUE=uvalue, $
  TUMBLE_CNT=tumble_cnt, TUMBLE_PERIOD=tumble_period

  ;This COMMON block holds the bitmaps and random number generator
  ;seed.
  COMMON CW_DICE_BLK, seed, faces

  ;Provide defaults for the keywords.
  IF ~ KEYWORD_SET(tumble_cnt) THEN tumble_cnt=10

  ;Guard against a nonsensical request.
  IF (tumble_cnt LT 1) THEN tumble_cnt=10

  ;Default tumble period in seconds.
  IF ~ KEYWORD_SET(tumble_period) THEN tumble_period=.05
  IF (tumble_period lt 0) THEN tumble_period=.05
  IF ~ KEYWORD_SET(uvalue) THEN uvalue=0

  ;Return to caller if an error occurs.
  ON_ERROR, 2
```

```
    ;Generate the die face bitmaps. The actual code for this is
    ;omitted here because it doesn't add much to the example, but
    ;it can be found in the CW_DICE.PRO file.
    faces=LONARR(192)

    ;Use RANDOMU to pick the initial value of the die unless the user
    ;provided one.
    IF(N_ELEMENTS(value) EQ 0) THEN value = FIX(6*RANDOMU(seed) + 1)

    ;Construct a state variable for this instance.
    state = { value:value, tumble_cnt:FIX(tumble_cnt), $
              tumble_period:tumble_period, remaining:0 }

    ;Create the base widget, passing the UVALUE through for the
    ;caller. Notice that we also register an event function and
    ;GET/SET value routines which will be called by WIDGET_CONTROL
    ;on our behalf.
    base = WIDGET_BASE(parent, UVALUE=uvalue, $
           EVENT_FUNC='CW_DICE_EVENT', $
           FUNC_GET_VALUE='CW_DICE_GET_VALUE', $
           PRO_SET_VALUE='CW_DICE_SET_VALUE')

    ;Create the die, setting its bitmap to the current value.
    die = WIDGET_BUTTON(base, VALUE=faces[*, *, value-1])

    ;Save the state in the first child's user value. Notice the
    ;use of the NO_COPY keyword for efficiency.
    WIDGET_CONTROL, WIDGET_INFO(base, /CHILD), $
      SET_UVALUE=state, /NO_COPY

    ;The result of a compound widget is always the ID of its topmost
    ;widget.
    RETURN, base

END
```

The above code makes reference to two routines named CW_DICE_SET_VAL and CW_DICE_GET_VAL. By using the FUNC_GET_VALUE and PRO_SET_VALUE keywords to WIDGET_BASE, WIDGET_CONTROL can call these routines whenever the user makes a WIDGET_CONTROL, SET_VALUE or GET_VALUE request:

```
;This is the SET_VALUE routine for CW_DICE. The number and type of
;the arguments is defined by WIDGET_CONTROL. Id is the widget ID of
;a CW_DICE, and value is the user's requested value.
PRO cw_dice_set_val, id, value

  COMMON CW_DICE_BLK, seed, faces
```

```
  ;Get the ID of the first child of the CW_DICE widget. This is
  ;where the state information is stored.
  stash = WIDGET_INFO(id, /CHILD)

  ;Get the state structure.
  WIDGET_CONTROL, stash, GET_UVALUE=state, /NO_COPY

  ;If the value is outside the range [1,6] then roll the die as if
  ;the user pressed the button.
  IF (value LT 1) OR (value GT 6) THEN BEGIN

    ;CW_DICE_ROLL rolls the dice. It's a separate function because
    ;our event handler also needs to use it.
    CW_DICE_ROLL, stash, state

  ENDIF ELSE BEGIN
    ;If the value is in the range [1,6] then simply set the die
    ;to that value without rolling.
    state.value=value

    ;Set the new bitmap on the button. We take advantage of the
    ;fact that stash must be the widget ID of the button widget,
    ;since the base only has one child.
    WIDGET_CONTROL, stash, SET_VALUE=faces[*,*, value-1]

  ENDELSE

  ;Restore the state in the child UVALUE.
  WIDGET_CONTROL, stash, SET_UVALUE=state, /NO_COPY

END


;This is the GET_VALUE routine for CW_DICE. The number and type of
;the arguments is defined by WIDGET_CONTROL. Id is the widget ID of
;a CW_DICE. The return value of this function must be the current
;value of the compound widget, as defined by that widget.
FUNCTION cw_dice_get_val, id

  ;Get the ID of the first child of the CW_DICE widget. This is
  ;where the state information is stored.
  stash = WIDGET_INFO(id, /CHILD)

  ;Get the state structure.
  WIDGET_CONTROL, stash, GET_UVALUE=state, /NO_COPY

  ;Get the current value from the state structure.
  ret = state.value
```

```
    ;Restore the state in the child UVALUE.
    WIDGET_CONTROL, stash, SET_UVALUE=state, /NO_COPY

    RETURN, ret

  END
```

CW_DICE_SET_VALUE makes reference to a procedure named CW_DICE_ROLL that does the actual dice rolling. Rolling is implemented as follows:

1. If this is the initial call to CW_DICE_ROLL, then pick the final value that will end up being displayed and enter this into the widget's state. Hence, WIDGET_CONTROL, /GET_VALUE reports the final value instead of one of the intermediate "tumble" values no matter when it is called.
2. If this is not the final tumble, pick a random intermediate value and display that. Then, make another timer event request for the next tumble.
3. If this is the final tumble, use the saved final value.
4. CW_DICE_ROLL works in cooperation with the event handler function for CW_DICE. Each timer event causes the event handler to be called and the event handler in turn calls CW_DICE_ROLL to process the next tumble.

```
;Roll the specified die. Dice is the widget ID of the button
;holding the bitmap, and state is the state as extracted from the
;CW_DICE UVALUE by the caller.
PRO cw_dice_roll, dice, state

  COMMON CW_DICE_BLK, seed, faces

  ;First time.
  IF (state.remaining EQ 0) THEN BEGIN

    ;Set the counter for the number of tumbles remaining.
    state.remaining = state.tumble_cnt

    ;Determine final value now.
    state.value = FIX(6*RANDOMU(seed)+1)

  ENDIF

  ;Last time.
  IF (state.remaining EQ 1) THEN BEGIN

    ;Use the previously-saved final result.
    value = state.value

  ;Not the last time.
```

```
    ENDIF ELSE BEGIN

      ;Generate an intermediate value.
      value = FIX(6 * RANDOMU(seed) + 1)

      ;Since this isn't the last tumble, make the next timer request.
      WIDGET_CONTROL, dice, TIMER=state.tumble_period

    ENDELSE

    ;Display the correct bitmap.
    WIDGET_CONTROL, dice, SET_VALUE=faces[*,*, value-1]

    ;Decrement tumble counter.
    state.remaining = state.remaining-1

  END
```

This leads us to the event handler function:

```
  FUNCTION cw_dice_event, event

    ;The primary use for the HANDLER field of event structures is to
    ;make finding the root of a compound widget easy.
    base = event.handler

    ;Get the ID of the first child of the CW_DICE widget. This is
    ;where the state information is stored.
    stash = WIDGET_INFO(base, /CHILD)

    ;Get the state structure.
    WIDGET_CONTROL, stash, GET_UVALUE=state, /NO_COPY

    ;Roll the die and display a new bitmap.
    CW_DICE_ROLL, stash, state

    ;This event handler expects to see button press events generated
    ;from a user action as well as TIMER events from CW_DICE_ROLL. We
    ;only want to issue events for the button presses. Even though
    ;the die still has several tumbles left, we know that the final
    ;value is in the state now.
    IF (TAG_NAMES(event, /STRUCTURE_NAME) NE 'WIDGET_TIMER') THEN $

      ;Create an event.
      ret = { CW_DICE_EVENT, ID:base, TOP:event.top, $
              HANDLER:0L, VALUE:state.value} $
    ELSE ret = 0
    ;By not returning an event structure, we cause the event to be
    ;swallowed by WIDGET_EVENT.
```

```
  ;Restore the state in the child UVALUE.
  WIDGET_CONTROL, stash, SET_UVALUE=state, /NO_COPY

  RETURN, ret

END
```

Using CW_DICE in a Widget Program

We can use CW_DICE to implement an application named XDICE. XDICE displays two dice as well as a “Roll” button. Pressing either die causes it to roll individually. Pressing the “Roll” button causes both dice to roll together. A text widget at the bottom displays the current value.

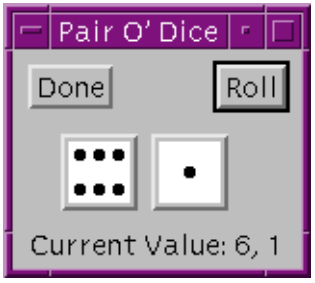


Figure 26-1: The XDICE Example Program

Note

`xdice.pro` can be found in the examples/widgets subdirectory of the IDL distribution. You can run the program from the IDL distribution by entering:

```
xdice
```

at the IDL command prompt. See “Running the Example Code” on page 738 if IDL does not run the program as expected. You should examine the files for additional details and comments not included here.

```
;Providing standard keywords usually found in other widget
;applications is a nice finishing touch. GROUP is easy to support
;since we just pass it to XMANAGER.
PRO xdice, GROUP=group

  ;Create the top-level base that holds everything else.
  base = WIDGET_BASE(/COLUMN, TITLE='Pair O'' Dice')

  ;A button group compound widget is used to implement the Done and
```

```
  ;Roll buttons. The SPACE keyword simply causes the buttons to be
  ;spread out from each other.
  bgroup = CW_BGROUP(base, ['Done', 'Roll'], /ROW, SPACE=50)

  ;Create a row base to hold the dice. XPAD moves the first die
  ;away from the left side of the application and helps center the
  ;dice.
  dice = WIDGET_BASE(base, /ROW, XPAD=20)

  ;The first die.
  d1 = CW_DICE(dice)

  ;The second die.
  d2 = CW_DICE(dice)

  ;We need the initial dice values to set the label appropriately.
  ;We could have specified initial values for the calls to CW_DICE
  ;above, but it seems better to let them be different on each
  ;invocation.
  WIDGET_CONTROL, d1, GET_VALUE=d1v
  WIDGET_CONTROL, d2, GET_VALUE=d2v

  ;Format the initial label text.
  str=STRING(FORMAT='("Current Value: ",I1,", ",I1)', d1v, d2v)

  ;This label is used to textually display the current dice values.
  label = WIDGET_LABEL(base, VALUE=str)

  ;Information that is needed in the event handler.
  state = { bgroup:bgroup, d1:d1, d2:d2, label:label }

  ;Save useful information in the base UVALUE, and realize the
  ;application.
  WIDGET_CONTROL, base, SET_UVALUE=state, /NO_COPY, /REALIZE

  ;Pass control to XMANAGER.
  XMANAGER, 'xdice', base, GROUP=group

END
```

The following event handler is called by XMANAGER to process events for the XDICE application:

```
PRO xdice_event, event

  ;Recover the state.
  WIDGET_CONTROL, event.top, GET_UVALUE=state, /NO_COPY

  ;Either the Done or Roll button was pressed.
```

```
  IF (event.ID EQ state.bgroup) THEN BEGIN

    ;The Done button.
    IF (event.VALUE EQ 0) THEN BEGIN

      WIDGET_CONTROL, /DESTROY, event.TOP   ;Destroy the
application.

      ;Return now to avoid trying to update the widget label we
      ;just destroyed.
      RETURN

    ;The Roll button.
    ENDIF ELSE BEGIN

      ;Roll the first die by asking for an out of range value.
      WIDGET_CONTROL, state.d1, SET_VALUE=-1

      ;Roll the second die.
      WIDGET_CONTROL, state.d2, SET_VALUE=-1

    ENDELSE
  ENDIF

  ;Get value of first die.
  WIDGET_CONTROL, state.d1, GET_VALUE=d1v

  ;Get value of second die.
  WIDGET_CONTROL, state.d2, GET_VALUE=d2v

  ;Format the initial label text.
  str = STRING(FORMAT='("Current Value: ",I1,", ",I1)', d1v, d2v)

  ;Update the label.
  WIDGET_CONTROL, state.label, SET_VALUE=str

  ;Restore the state.
  WIDGET_CONTROL, event.TOP, SET_UVALUE=state, /NO_COPY

END
```

Debugging Widget Applications

In addition to the “normal” debugging tasks associated with any IDL program, widget applications also require you to debug errors in the widget event loop. If your widget application experiences errors in an event handling routine, keep the following points in mind:

- By default, XMANAGER catches errors and continues processing (see “CATCH” in the *IDL Reference Guide* manual). If you are using XMANAGER to manage your widget application (as in most cases you should), calling XMANAGER with CATCH=0 will cause XMANAGER to halt when it encounters an error.

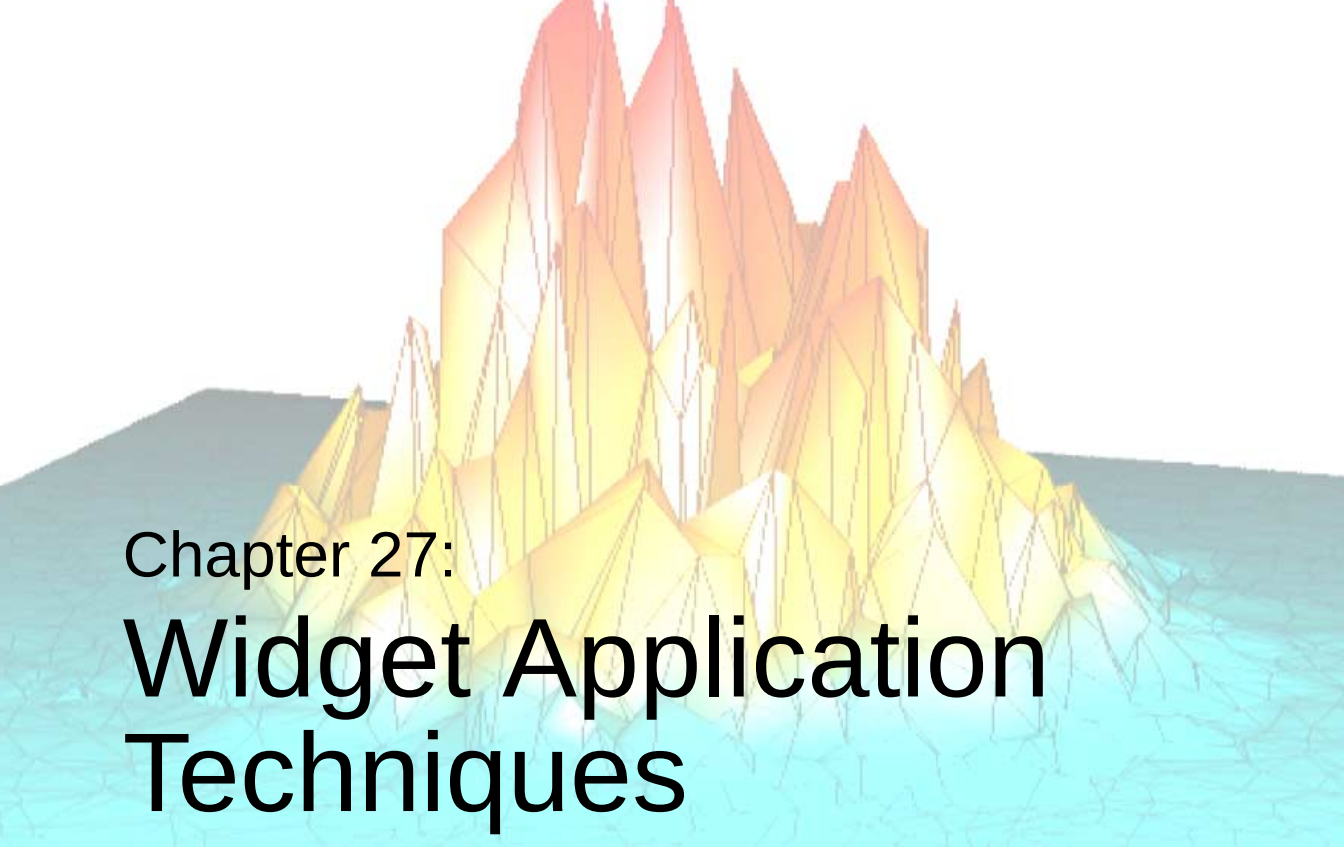
  Setting CATCH=0 is useful during debugging, but finished programs should run with the default setting (CATCH=1) and refrain from setting it explicitly.

- CATCH is only effective if XMANAGER is blocking to dispatch errors. During debugging, make sure to call XMANAGER with NO_BLOCK=0 (the default).

  Setting NO_BLOCK=0 is useful during debugging, but in many cases you will want your finished program to set NO_BLOCK=1 in order to allow other widget programs (and the IDL command line) to remain active while your application is running.

If a widget application stops responding, you can restart event processing by doing the following:

1. Enter `RETALL` at the IDL prompt to return to the main program level.
2. Optionally, modify the code to fix the error and re-compile.
3. If one or more of the applications you are running blocks the active command line, enter `XMANAGER` at the IDL prompt in order to have it resume processing events in the blocking mode. If all applications have NO_BLOCK=1 set, a call to `XMANAGER` will immediately return, and can be safely omitted.



Chapter 27: Widget Application Techniques

The following topics are covered in this chapter:

Working with Widget Events

Widget events and the process of establishing a widget event loop for your application are described in "Widget Event Processing" in Chapter 26. This section discusses additional topics that may be useful when creating event-driven applications, including:

- Interrupting the Event Loop
- Identifying Widget Type from an Event
- Keyboard Focus Events
- Timer Events
- Tracking Events
- Context Menu Events

Interrupting the Event Loop

Beginning with IDL version 5, IDL has the ability to process commands from the IDL command line while simultaneously processing widget events. This means that the IDL command line will remain active even when widget applications are running.

It is possible to interrupt the event function by sending the interrupt character (Control-C). However, you may find that even after sending the interrupt character, IDL does not immediately interrupt the event loop. IDL will interrupt the process that is "on top"—that is, if several applications are running at once, the interrupt will be handled by the first application to receive it.

If your widget application is the only active application, and sending the interrupt does not cause it to break, move the mouse cursor across (or click on) one of the widgets.

This works because when IDL is in the event function, it only checks for the interrupt between event notifications from the window system. Such events do not necessarily translate one-to-one into IDL widget events because the window system typically generates a large number of events related to the window system's operation that IDL quietly handles. Moving the mouse cursor across the widgets typically generates some of these events which gives IDL a chance to notice the interrupt and act on it.

Identifying Widget Type from an Event

Given a widget event structure, often you need to know what type of widget generated it without having to match the widget ID in the event structure to all the current widgets. This information is available by specifying the STRUCTURE_NAME keyword to the TAG_NAMES function:

```
PRINT, 'Event structure type: ', TAG_NAMES(EVENT, /STRUCTURE_NAME)
```

This works because each widget type generates a different event structure. The event structure generated by a given widget type is documented in the description of the widget creation function in the *IDL Reference Guide*.

When using this technique, be aware that although all the basic widgets use named structures for their events, many compound widgets return anonymous structures. This technique does not work well in that case because anonymous structures lack a recognizable name.

An alternative technique involves using the TYPE keyword to the WIDGET_INFO function. This method is useful when the widget event name does not specify the widget from which the event originated. Timer events are an example; although the events originate from a widget, the event structure's name is WIDGET_TIMER. The following statement checks to see if the event is a timer event and, if it is, prints the type code of the widget that generated the event.

```
IF ((TAG_NAMES(EVENT, /STRUCTURE) EQ 'WIDGET_TIMER') THEN $
  PRINT, WIDGET_INFO(EVENT.ID, /TYPE)
```

Such a check would be useful if a given widget could generate *either* a timer event or a "normal" event, and you wanted to differentiate between the two.

Note

Always check for a distinct type of widget event. RSI will continue to add new widgets with new event structures, so it is important not to make assumptions about the contents of a random widget event structure. The structure of existing widget events will remain stable, (although new fields may be added) so checking for a particular type of widget event will always work.

Keyboard Focus Events

Base, table, and text widgets can be set to generate *keyboard focus events*. Generating and examining keyboard focus events allows you to determine when a given widget has either *gained* or *lost* the keyboard focus—that is, when it is brought to the foreground or when it is covered by another window.

Set the KBRD_FOCUS_EVENTS keyword to WIDGET_BASE, WIDGET_TABLE, or WIDGET_TEXT to generate keyboard focus events. (You can also modify an existing base, table, or text widget to generate keyboard focus events using the KBRD_FOCUS_EVENTS keyword to WIDGET_CONTROL.) You can then use your event-handling procedure to cache the widget ID of the last widget (with keyboard focus events enabled) to have the keyboard focus. One situation where this is useful is when you have an application menu (created with the MBAR keyword to WIDGET_BASE) and you wish to perform an action in a text widget based on the menu item selected. Although the event generated by the user's menu selection has the *menu's* base as its top-level widget ID, if you generate and track keyboard focus events for the text widget, you can "remember" which widget the action triggered by the menu selection should affect. Note that in this example, keyboard focus events are *not* generated for the menubar's base.

Timer Events

In addition to the normal widget events discussed previously, IDL allows the user to make *timer event* requests by using the TIMER keyword. Such events are useful in many applications that are time dependent, such as animation. The syntax for making such a request is:

```
WIDGET_CONTROL, Widget_Id, TIMER=interval_in_seconds
```

Widget_Id can be the ID of any type of widget. When such a request is made, IDL generates a timer request after the requested time interval has passed. Timer events consist of a structure with only the standard three fields — no additional information is provided.

It is up to the programmer to differentiate between a normal event and a timer event for a given widget. The usual way to solve this problem is to make timer requests for widgets that do not otherwise generate events, such as base or label widgets.

Each timer request causes a single event to be generated. To generate a steady stream of timer events, you must make a new timer request in the event handler routine each time a timer event is delivered. The following example demonstrates how to check for a timer event and generate a new timer event each time a timer event occurs:

```
PRO timer_example_event, ev

  WIDGET_CONTROL, ev.ID, GET_UVALUE=uval
  IF (TAG_NAMES(ev, /STRUCTURE_NAME) EQ 'WIDGET_TIMER') THEN BEGIN
    PRINT, 'Timer Fired'
    WIDGET_CONTROL, ev.TOP, TIMER=2
  ENDIF
```

```
  CASE uval OF
    'timer' : BEGIN
      WIDGET_CONTROL, ev.TOP, TIMER=2
      END
    'exit' : WIDGET_CONTROL, ev.TOP, /DESTROY
  ELSE:
  ENDCASE

END

PRO timer_example
  base = WIDGET_BASE(/COLUMN, UVALUE='base')
  b1 = WIDGET_BUTTON(base, VALUE='Fire event', UVALUE='timer')
  b2 = WIDGET_BUTTON(base, VALUE='Exit', UVALUE='exit')
  WIDGET_CONTROL, base, /REALIZE
  XMANAGER, 'timer_example', base, /NO_BLOCK
END
```

See “Draw Widget Example” on page 820 for a larger example using timer events.

Tracking Events

Tracking events allow you to determine when the mouse pointer has entered or left the area of the computer screen covered by a given widget. You can use tracking events to allow your interface to react as the user moves the mouse pointer over different interface elements. Tracking events are generated for a widget when the widget creation routine is called with the TRACKING_EVENTS keyword set.

The event structure of a tracking event includes a field named ENTER that contains a 1 (one) if the mouse pointer entered the region covered by the widget, or 0 (zero) if the mouse pointer left the region covered by the widget. The following example demonstrates how to check for tracking events and modify the value of a button widget when the mouse cursor is positioned over it.

```
PRO tracking_demo_event, event
  IF (TAG_NAMES(event, /STRUCTURE_NAME) EQ 'WIDGET_TRACKING') $
  THEN BEGIN
    IF (event.ENTER EQ 1) THEN BEGIN
      WIDGET_CONTROL, event.ID, SET_VALUE='Press to Quit'
    ENDIF ELSE BEGIN
      WIDGET_CONTROL, event.ID, $
        SET_VALUE='What does this button do?'
    ENDELSE
  ENDIF ELSE BEGIN
    WIDGET_CONTROL, event.TOP, /DESTROY
  ENDELSE
END
```

```
PRO tracking_demo
  base = WIDGET_BASE(/COLUMN)
  button = WIDGET_BUTTON(base, $
    VALUE='What does this button do?', /TRACKING_EVENTS)
  WIDGET_CONTROL, base, /REALIZE
  XMANAGER, 'tracking_demo', base
END
```

Context Menu Events

Base, list, text, and tree widgets can be set to generate *context menu events*. Generating and examining context menu events allows you to determine when the user has clicked the right-hand mouse button over a given widget, which in turn allows you to display a “context menu.” (Draw widgets can also generate events when the right-hand mouse button is clicked, using the general BUTTON_EVENTS mechanism.) See “Context Menu Events” on page 786 for a detailed description.

Using Multiple Widget Hierarchies

Using widgets, you can create IDL applications with graphical user interfaces. Although widget applications are running “inside” IDL, a well-designed program can behave and appear just like a stand-alone application.

While a simple application may consist of a single widget hierarchy headed by a single top-level base widget, more complex applications can include multiple widget hierarchies, each with their own top-level base. Widget applications that include multiple widget hierarchies consist of a *group* of top-level base widgets organized hierarchically. The individual widgets that make up the widget application’s interface have as their parent widget either one of the top-level bases or a base that is a child of one of the top-level bases.

Groups of widgets are defined by setting the GROUP_LEADER keyword when creating the widget. Group membership controls how and when widgets are iconized, which layer they appear in, and when they are destroyed.

Figure 27-1 depicts a widget application group hierarchy consisting of six top-level bases in three groups: base 1 leads all six bases, base 2 leads bases 4 and 5, and base 3 leads base 6. What does this mean? Operations like inconization or destruction that affect base 2 also affect bases 4 and 5. Operations that affect base 3 also affect base 6. Operations that affect base 1 affect all six bases—that is, a group includes not only those bases that explicitly claim one base as their leader, but also all bases *led by* those member bases.

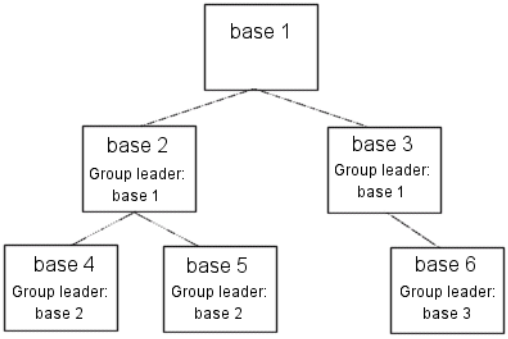


Figure 27-1: A widget application group hierarchy with six top-level bases.

The following IDL commands would create this hierarchy:

```
base1 = WIDGET_BASE()
base2 = WIDGET_BASE(GROUP_LEADER=base1)
base3 = WIDGET_BASE(GROUP_LEADER=base1)
base4 = WIDGET_BASE(GROUP_LEADER=base2)
base5 = WIDGET_BASE(GROUP_LEADER=base2)
base6 = WIDGET_BASE(GROUP_LEADER=base3)
```

Widget Group Behaviors

Groups of widgets are displayed and destroyed according to the following principles:

Iconization

Bases and groups of bases can be *iconized* (or *minimized*) by clicking the system minimize control. When a group leader is iconized, all members of the group are minimized as well.

Layering

Layering is the process by which groups of widgets seem to share the same plane on the display screen. Within a layer on the screen, widgets have a *Z-order*, or front-to-back order, that defines which widgets appear to be on top of other widgets.

All widgets within a group hierarchy share the same layer—that is, when one group member has the input focus, all members of the group hierarchy are displayed in a layer that appears in front of all other groups or applications. Within the layer, the widgets can have an arbitrary Z-order, determined by the programmer.

Destruction

When a group leader widget is destroyed, either programmatically or by clicking on the system “close” button, all members of the group are destroyed as well.

See “Iconizing, Layering, and Destroying Groups of Top-Level Bases” under “WIDGET_BASE” in the *IDL Reference Guide* manual for detailed information on how group membership defines widget behavior on different platforms.

Floating bases

Top-level base widgets created with the FLOATING keyword set will *float* above their group leaders, even though they share the same layer. Floating bases and their group leaders are iconized in a single icon (on platforms where iconization is possible). Floating bases are destroyed when their group leaders are destroyed.

Modal bases

Top-level base widgets created with the MODAL keyword will float above their group leaders, and will suspend processing in the widget application until they are dismissed. (*Dialogs* are generally modal.) Modal bases cannot be iconized, and on some platforms other bases cannot be moved or iconized while the modal dialog is present. Modal bases cannot have scroll bars or menubars.

Menubars

Widget applications can have an application-specific menubar, created by the MBAR keyword to WIDGET_BASE. Menus and menubars are discussed in detail in "Creating Menus" on page 790.

Creating Menus

Menus allow a user to select one or more options from a list of options. IDL widgets allow you to build a number of different types of menus for your widget application.

This section discusses the following different types of menus:

- Button Groups
- Lists
- Pulldown Menus
- Menus on Top-Level Bases
- Context-Sensitive Menus

Button Groups

One approach to menu creation is to build an array of buttons. With a button menu, all options are visible to the user all the time. To create a button menu, do the following:

1. Call the WIDGET_BASE function to create a base to hold the buttons. Use the COLUMN and ROW keywords to determine the layout of the buttons.
2. Call the WIDGET_BUTTON function once for each button to be added to the base created in the previous step.

Because menus of buttons are common, IDL provides a compound widget named CW_BGROUP to create them. Using CW_BGROUP rather than a series of calls to WIDGET_BUTTON simplifies creation of a menu of buttons and also simplifies event handling by providing a single event structure for the group of buttons. For example, the following IDL statements create a button menu with five choices:

```
values = ['One', 'Two', 'Three', 'Four', 'Five']
base = WIDGET_BASE()
bgroup = CW_BGROUP(base, values, /COLUMN)
WIDGET_CONTROL, base, /REALIZE
```

In this example, one call to CW_BGROUP replaces five calls to WIDGET_BUTTON.

Exclusive or Nonexclusive Buttons

Buttons in button groups normally act as independent entities, returning a selection event (a one in the select field of the event structure) or similar value when pressed. Groups of buttons can also be made to act in concert, as either exclusive or non-

exclusive groups. In contrast to normal button groups, both exclusive and non-exclusive groups display which buttons have been selected.

Exclusive button groups allow only one button to be selected at a given time. Clicking on an unselected button deselects any previously-selected buttons. *Non-exclusive* button groups allow any number of buttons to be selected at the same time. Clicking on the same button repeatedly selects and deselects that button.

The following code creates three button groups. The first group is a “normal” button group as created in the previous example. The next is an exclusive group, and the third is a non-exclusive group.

```
values = ['One', 'Two', 'Three', 'Four', 'Five']
base = WIDGET_BASE(/ROW)
bgroup1 = CW_BGROUP(base, values, /COLUMN, $
  LABEL_TOP='Normal', /FRAME)
bgroup2 = CW_BGROUP(base, values, /COLUMN, /EXCLUSIVE, $
  LABEL_TOP='Exclusive', /FRAME)
bgroup3 = CW_BGROUP(base, values, /COLUMN, /NONEXCLUSIVE, $
  LABEL_TOP='Nonexclusive', /FRAME)
WIDGET_CONTROL, base, /REALIZE
```

The widget created by this code is shown in the following figure:

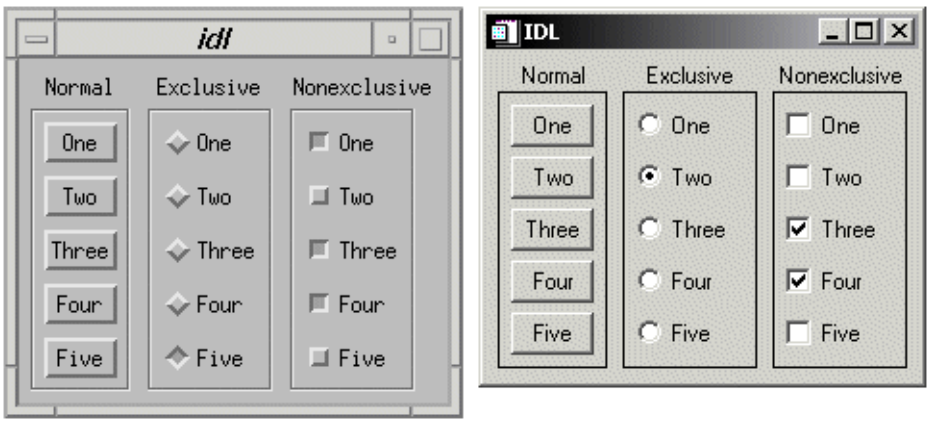


Figure 27-2: Normal Menus (left), Exclusive Menus (center) and Non-exclusive Menus (right)

Lists

A second approach to menu creation is to provide the user with a list of options in the form of a scrolling or drop-down list. A scrolling list is always displayed, although it may not show all items in the list at all times. A drop-down list shows only the selected item until the user clicks on the list, at which time it displays the entire list. Both lists allow only a single selection at a time.

The following example code uses the WIDGET_LIST and WIDGET_DROPLIST functions to create two menus of five items each. While both lists contain five items, the scrolling list displays only three at a time, because we specify this with the YSIZE keyword.

```
values = ['One', 'Two', 'Three', 'Four', 'Five']
base = WIDGET_BASE(/ROW)
list = WIDGET_LIST(base, VALUE=values, YSIZE=3)
drop = WIDGET_DROPLIST(base, VALUE=values)
WIDGET_CONTROL, base, /REALIZE
```

The widget created by this code is shown in the following figure:

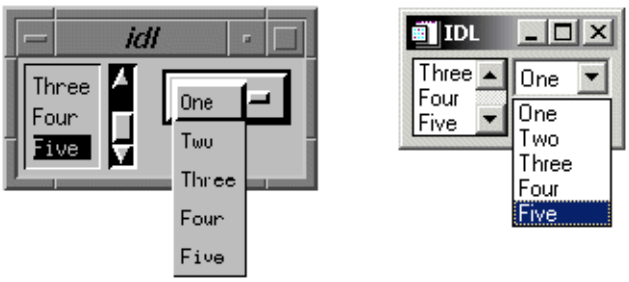


Figure 27-3: Scrolling and drop-down lists.

Pulldown Menus

A third approach to menu creation involves menus that appear as a single button until the user selects the menu, at which time the menu pops up to display the list of possible selections. Buttons in such a pulldown menu can activate other pulldown menus to any desired depth. The method for creating a pulldown menu is as follows:

1. The topmost element of any pulldown menu is a button, created with the MENU keyword to the WIDGET_BUTTON function.

2. The top-level button has one or more child widget buttons attached. (That is, one or more buttons specify the first button's widget ID as their "parent.") Each button can either be used as is, in which case pressing it causes an event to be generated, or it can be created with the MENU keyword and have further child widget buttons attached to it. If it has child widgets, pushing it causes a pulldown menu containing the child buttons to pop into view.
3. Menu buttons can be the parent of other buttons to any desired depth.

Because pulldown menus are common, IDL provides a compound widget named CW_PDMENU to create them. Using CW_PDMENU rather than a series of calls to WIDGET_BUTTON simplifies creation of a pulldown menu in the same way the CW_BGROUP simplifies the creation of button menus.

The following example uses CW_PDMENU to create a pulldown menu. First, we create an array of anonymous structures to contain the menu descriptions.

```
desc = REPLICATE({ flags:0, name:'' }, 6)
```

The desc array contains six copies of the empty structure. Each structure has two fields: flags and name. Next, we populate these fields with values:

```
desc.flags = [ 1, 0, 1, 0, 2, 2 ]
desc.name = [ 'Operations', 'Predefined', 'Interpolate', $
              'Linear', 'Spline', 'Quit' ]
```

The value of the flags field specifies the role of each button. In this example, the first and third buttons start a new sub-menu (values are 1), the second and fourth buttons are plain buttons with no other role (values are 0), and the last two buttons end the current sub-menu and return to the previous level (values are 2). The value of the name field is the value (or label) of the button at each level.

```
base = WIDGET_BASE()
menu = CW_PDMENU(base, desc)
WIDGET_CONTROL, base, /REALIZE
```

The format of the menu description used by CW_PDMENU in the above example requires some explanation. CW_PDMENU views a menu as consisting of a series of buttons, each of which can optionally lead to a sub-menu. The description of each button consists of a structure supplying its name and a flag field that tells what kind of button it is (starts a new sub-menu, ends the current sub-menu, or a plain button within the current sub-menu). The description of the complete menu consists of an

array of such structures corresponding to the flattened menu. Compare the description used in the code above with the result shown in the following figure.

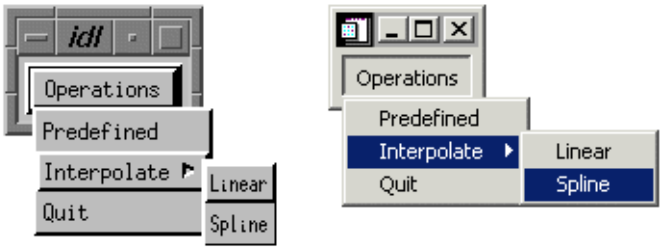


Figure 27-4: Pulldown menus created with CW_PDMENU.

Menus on Top-Level Bases

A fourth approach to providing menus in your widget application is to attach the menus directly to the top-level base widget. Menus attached to a top-level base widget are created just like pulldown menus created from button widgets, but they do not appear as buttons. Menus created in this way are children of a special sub-base of the top-level base, created by specifying the MBAR keyword when the top-level base is created.

For example, the following code creates a top-level base widget and attaches a menu titled MENU1 to it. MENU1 contains the choices ONE, TWO, and THREE.

```
base = WIDGET_BASE(MBAR=bar)
menu1 = WIDGET_BUTTON(bar, VALUE='MENU1', /MENU)
button1 = WIDGET_BUTTON(menu1, VALUE='ONE')
button2 = WIDGET_BUTTON(menu1, VALUE='TWO')
button3 = WIDGET_BUTTON(menu1, VALUE='THREE')
draw = WIDGET_DRAW(base, XSIZE=100, YSIZE=100)
WIDGET_CONTROL, base, /REALIZE
```

The resulting widget is shown in the following figure:

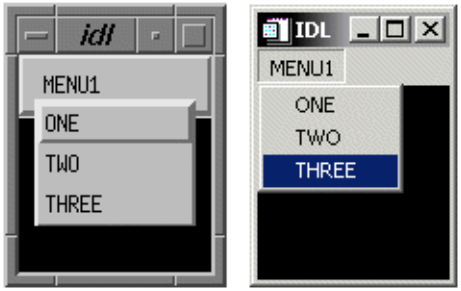


Figure 27-5: Menus attached to a top-level base.

Context-Sensitive Menus

Context-sensitive menus (also referred to as *context menus* or *pop-up menus*) are hidden until a user performs an action to display the menu. When summoned, the appearance of a context menu is similar to that of a menu created in a floating, modal base. The behavior of a context menu is the same as that of a menu on a menu bar; when the user clicks one of the menu's buttons, a button event is generated and the menu is dismissed. If the user clicks outside the context menu, it is dismissed without generating any events.

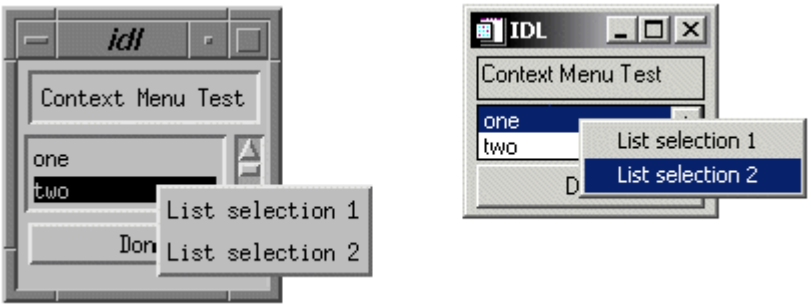


Figure 27-6: Widget Context Menus.

By convention, context-sensitive menus in IDL widget applications are displayed when the user clicks the right mouse button. Five IDL widget primitives — base,

draw, list, text, and tree widgets — can be configured to generate events when this occurs. The mechanism used to generate right mouse button events is different for draw widgets than for the other types; these differences are discussed below.

Note

While it is customary to display context-sensitive menus when the user clicks the right mouse button, IDL's mechanism for displaying the menus is quite general, and can be invoked under many circumstances. Examples in this section will discuss the common usage.

To create a context-sensitive menu in a widget application, do the following:

1. Create a Context Menu
2. Generate and Handle Context Events
3. Display the Context Menu
4. Process Button Events

Create a Context Menu

Context menus are contained within a special base widget created with the CONTEXT_MENU keyword. A base widget used as the base for a context menu must have as its parent widget one of the following widget types:

- Base widget
- Draw widget
- List widget
- Text widget
- Tree widget

The process for creating a context menu is similar to that for creating a menu for a top-level base (a menubar). Create menu entries on the base widget using the WIDGET_BUTTON function. Context menu entries can display sub-menus (using the MENU keyword to WIDGET_BUTTON or the CW_PDMENU compound widget) or appear as separators (using the SEPARATOR keyword to WIDGET_BUTTON).

The following code snippet illustrates a very simple context menu attached to a base widget:

```
topLevelBase = WIDGET_BASE(/CONTEXT_EVENTS)
contextBase = WIDGET_BASE(topLevelBase, /CONTEXT_MENU)
button1 = WIDGET_BUTTON(contextBase, VALUE='First button')
```

```
button2 = WIDGET_BUTTON(contextBase, VALUE='Second button')
```

Generate and Handle Context Events

Generating Right Mouse Button Events

In order to display the context menu at the appropriate time, the widget that serves as the parent for the context menu base must be configured to generate an event when the user clicks the right mouse button over that widget. For base, list, text, and tree widgets, this is accomplished by setting the CONTEXT_EVENTS keyword when creating the widget, or by enabling context events by setting the CONTEXT_EVENTS keyword to WIDGET_CONTROL. When a user clicks the right mouse button over an appropriately configured base, list, text, or tree widget, a widget event with the following structure is generated:

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

The first three fields are the standard fields found in every widget event. The X and Y fields give the device coordinates at which the event occurred, measured from the upper left corner of the base widget.

For draw widgets, button events are handled differently. Set the BUTTON_EVENTS keyword to WIDGET_DRAW (or the DRAW_BUTTON_EVENTS keyword to WIDGET_CONTROL) to generate widget events with the following structure:

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

The first three fields are the standard fields found in every widget event. The X and Y fields give the device coordinates at which the event occurred, measured from the lower left corner of the drawing area. PRESS and RELEASE are bitmasks that represent which of the left, center, or right mouse button was pressed: that is, a value of 1 (one) represents the left button, 2 represents the middle button, and 4 represents the right button. (See “Widget Events Returned by Draw Widgets” in the *IDL Reference Guide* manual for a complete description of the WIDGET_DRAW event structure.)

Detecting Right Mouse Button Events

Once the parent widget of your context menu is configured to generate events when the user clicks the right mouse button, you must detect the events in your event handler routine. For base, list, text, and tree widgets, your event handler should examine the event structure name to determine the type of event; if the event is of type WIDGET_CONTEXT, you know that the right mouse button was pressed.

To detect a right mouse button click in a base, list, text, or tree widget (with context events enabled), use the following test:

```
IF (TAG_NAMES(event, /STRUCTURE_NAME) EQ 'WIDGET_CONTEXT') THEN
BEGIN
  ; process event here
ENDIF
```

For draw widgets, your event handler should examine the WIDGET_DRAW event structure; if the value of the RELEASE field is equal to four, you know that the right mouse button was pressed and released.

To detect a right mouse button click in a draw widget (with button events enabled), use the following test:

```
IF (event.release EQ 4) THEN BEGIN
  ; process event here
ENDIF
```

Note that in a complex widget application, your event handler may first need to determine whether the event came from a draw widget. In this case, you may need a test that looks like this:

```
IF (TAG_NAMES(event, /STRUCTURE_NAME) EQ 'WIDGET_DRAW') THEN BEGIN
  IF (event.release EQ 4) THEN BEGIN
    ; process event here
  ENDIF
ENDIF
```

Display the Context Menu

When your event handler routine detects a right mouse button click, use the WIDGET_DISPLAYCONTEXTMENU procedure to display the context menu. This routine takes as its parameters the widget ID of the widget for which the context menu is to be displayed, the X and Y coordinates at which the menu should be displayed, and the widget ID of the context menu base widget that holds the context menu. See "WIDGET_DISPLAYCONTEXTMENU" in the *IDL Reference Guide* manual for additional information.

In all cases, the ID field of the event structure generated by the right mouse button click contains the widget ID of the widget whose context menu is to be displayed. Similarly, the event structure contains the location of the mouse click in the X and Y fields; in most cases, this is where you will want to display the context menu.

The following code fragment would display a context menu held in a base widget whose widget ID is `contextBase` at the location of the user's right mouse click:

```
WIDGET_DISPLAYCONTEXTMENU, event.ID, event.X, $
  event.Y, contextBase
```

In a simple application with only one context menu, you know the widget ID of the context menu base widget to be displayed. In a real application, however, it is likely

that more than one context menu exists. See “Determining Which Context Menu to Display”, below, for tips on dealing with multiple context menus.

Process Button Events

Once the context menu is displayed, processing events that flow from it is the same as processing events from any other menu. The individual buttons that make up the menu can have event handler routines associated with them; these routines are then invoked when the user clicks on one of the menu buttons. See the Example below for a simple illustration of menu button event processing.

Determining Which Context Menu to Display

In a real application, you may have multiple context menus available to display when the user right-clicks on different portions of the user interface. One way to handle this situation is to have your event handler keep track of which context menu should be displayed for each widget. Consider a widget hierarchy that contains a text widget and a list widget, both of which have associated context menus:

```
topLevelBase = WIDGET_BASE(/COLUMN, XSIZE = 120, YSIZE = 80)
wText = WIDGET_TEXT(topLevelBase, VALUE="Context Menu Test", $
  /CONTEXT_EVENTS)
wList = WIDGET_LIST(topLevelBase, VALUE=['one','two', 'three'], $
  /CONTEXT_EVENTS)
contextBase1 = WIDGET_BASE(wText,  /CONTEXT_MENU, $
  UNAME="tContextMenu")
contextBase2 = WIDGET_BASE(wList,  /CONTEXT_MENU, $
  UNAME="lContextMenu")
```

Now the application’s event handler, after detecting a right mouse button click with the

```
IF (TAG_NAMES(event, /STRUCTURE_NAME) EQ 'WIDGET_CONTEXT')
```

test, must somehow determine whether the user had clicked on the text widget or the list widget. To make this determination, you could use the WIDGET_INFO function to search the widget hierarchy starting with the widget at the top of the event structure for a widget with the correct UNAME value:

```
IF (WIDGET_INFO(event.id, FIND_BY_UNAME = 'tContextMenu') GT 0) $
  THEN BEGIN
  WIDGET_DISPLAYCONTEXTMENU, event.id, event.x, event.y, $
    WIDGET_INFO(event.id, FIND_BY_UNAME = 'tContextMenu')
ENDIF
IF (WIDGET_INFO(event.id, FIND_BY_UNAME = 'lContextMenu') GT 0) $
  THEN BEGIN
  WIDGET_DISPLAYCONTEXTMENU, event.id, event.x, event.y, $
    WIDGET_INFO(event.id, FIND_BY_UNAME = 'lContextMenu')
```

```
ENDIF
```

While this method will always work, it may involve a substantial amount of code, and must search the widget hierarchy multiple times to find the widget ID of the base for the context menu. If, however, your application has at most one context menu for each base, draw, list, or text widget, you can streamline the code significantly by using a common UNAME value for all of the context menus. For example, if the definitions of the context menu bases change to this:

```
contextBase1 = WIDGET_BASE(wText,  /CONTEXT_MENU, $
  UNAME="contextMenu")
contextBase2 = WIDGET_BASE(wList,  /CONTEXT_MENU, $
  UNAME="contextMenu")
```

then the code detecting and displaying the context menu becomes:

```
contextBase = WIDGET_INFO(event.ID, FIND_BY_UNAME = 'contextMenu')

WIDGET_DISPLAYCONTEXTMENU, event.ID, event.X, $
  event.Y, contextBase
```

Since the context menu base is a child of the text or list widget, the call to WIDGET_INFO finds the appropriate base by searching for the UNAME value “contextMenu”, starting at the widget specified by `event.ID`.

Example

The following example defines a simple application with two context menus, one each for a list widget and a text widget. When a menu item on one of the context menus is selected, IDL prints an informational message.

Note

This example is included in the file `context_menu_example.pro` in the `examples/widgets` subdirectory of the IDL distribution. You can either open the file in an IDL editor window and compile and run the code using items on the **Run** menu, or simply enter

```
context_menu_example
```

at the IDL command prompt. See “Running the Example Code” on page 738 if IDL does not run the program as expected.

```
; Define event handlers for context menu button events
PRO CME_11Event, event
  PRINT, ' '
  PRINT, 'Context Menu 1 Selection 1 pressed'
END

PRO CME_12Event, event
  PRINT, ' '
```

```
  PRINT, 'Context Menu 1 Selection 2 pressed'
END

PRO CME_21Event, event
  PRINT, ' '
  PRINT, 'Context Menu 2 Selection 1 pressed'
END

PRO CME_22Event, event
  PRINT, ' '
  PRINT, 'Context Menu 2 Selection 2 pressed'
END

; Define main event handler
PRO context_menu_example_event, event

  ; Test for context menu events
  IF (TAG_NAMES(event, /STRUCTURE_NAME) EQ 'WIDGET_CONTEXT') $
    THEN BEGIN
    ; Obtain the widget ID of the context menu base.
    contextBase = WIDGET_INFO(event.ID, $
      FIND_BY_UNAME = 'contextMenu')
    ; Display the context menu and send its events to the
    ; other event handler routines.
    WIDGET_DISPLAYCONTEXTMENU, event.ID, event.X, $
      event.Y, contextBase
  ENDIF

  ; Test for button event to end application
  IF (TAG_NAMES(event, /STRUCTURE_NAME) EQ 'WIDGET_BUTTON') $
    THEN BEGIN
    WIDGET_CONTROL, event.top, /DESTROY
  ENDIF

END

; Create GUI
PRO context_menu_example
  topLevelBase = WIDGET_BASE(/COLUMN, XSIZE = 120, YSIZE = 80)
  wText = WIDGET_TEXT(topLevelBase, VALUE="Context Menu Test", $
    /CONTEXT_EVENTS, /ALL_EVENTS)
  wList = WIDGET_LIST(topLevelBase, VALUE=['one','two','three'], $
    /CONTEXT_EVENTS)
  contextBase1 = WIDGET_BASE(wText,  /CONTEXT_MENU, $
    UNAME="contextMenu")
  contextBase2 = WIDGET_BASE(wList,  /CONTEXT_MENU, $
    UNAME="contextMenu")
  doneButton = WIDGET_BUTTON(topLevelBase, VALUE="Done")
```

```
    ; Initialize the buttons of the context menus.
    cb11 = WIDGET_BUTTON(contextBase1, VALUE = 'Text selection 1', $
      EVENT_PRO = 'CME_11Event')
    cb12 = WIDGET_BUTTON(contextBase1, VALUE = 'Text selection 2', $
      EVENT_PRO = 'CME_12Event')
    cb21 = WIDGET_BUTTON(contextBase2, VALUE = 'List selection 1', $
      EVENT_PRO = 'CME_21Event')
    cb22 = WIDGET_BUTTON(contextBase2, VALUE = 'List selection 2', $
      EVENT_PRO = 'CME_22Event')

    ; Display the GUI.
    WIDGET_CONTROL, topLevelBase, /REALIZE

    ; Handle the events from the GUI.
    XMANAGER, 'context_menu_example', topLevelBase

END
```

Additional Examples

The following additional examples using the context menu in various situations can be found in the `widgets` subdirectory of the `examples` subdirectory of the IDL distribution:

- context_tlbase_example.pro
- context_draw_example.pro
- context_list_example.pro
- context_text_example.pro

Widget Sizing

This section explains how IDL widgets size themselves, widget geometry concepts, and how to explicitly size and position widgets.

Widget Geometry Terms and Concepts

Widget size and layout is determined by many interrelated factors. In the following discussion, the following terms are used:

- *Geometry*: The size and position of a widget.
- *Natural Size*: The natural, or implicit, size of a widget is the size a widget has if no external constraints are placed on it. For example, a label widget has a natural size that is determined by the size of the text it is displaying and space for margins. These values are influenced by such things as the size of the font being displayed and characteristics of the low-level (i.e., operating-system level) widget or control used to implement the IDL widget.
- *Explicit Size*: The explicit, or user-specified, size of a widget is the size set when an IDL programmer specifies one of the size keywords to an IDL widget creation function or WIDGET_CONTROL.

How Widget Geometry is Determined

IDL uses the following rules to determine the geometry of a widget:

- The explicit size of a widget, if one is specified, takes precedence over the natural size. That is, the user-specified size is used if available.
- If an explicit size is not specified, the natural size of the widget—at the time the widget is realized—is used. Once realized, the size of a widget *does not automatically change* when the value of the widget changes, unless the widget's dynamic resize property has been set. Dynamic resizing is discussed in more detail below. Note that any realized widget can be made to change its size by calling WIDGET_CONTROL with any of the sizing keywords.
- Children of a "bulletin board" base (i.e., a base that was created without setting the COLUMN or ROW keywords) have an offset of (0,0) unless an offset is explicitly specified via the XOFFSET or YOFFSET keywords.
- The offset keywords to widgets that are children of ROW or COLUMN bases are ignored, and IDL calculates the offsets to lay the children out in a grid. This

calculation can be influenced by setting any of the ALIGN or BASE_ALIGN keywords when the widgets are created.

Dynamic Resizing

Realized widgets, by default, do not automatically resize themselves when their values change. This is true whether the widget was created with an explicit size or the widget was allowed to size itself naturally. This behavior makes it easy to create widget layouts that don't change size too frequently or "flicker" due to small changes in a widget's natural size.

This default behavior can be changed for label, button, and droplist widgets. Set the DYNAMIC_RESIZE keyword to WIDGET_LABEL, WIDGET_BUTTON, or WIDGET_DROPLIST to make a widget that automatically resizes itself when its value changes. Note that the XSIZE and YSIZE keywords should not be used with DYNAMIC_RESIZE. Setting explicit sizing values overrides the dynamic resize property and creates a widget that *will not* resize itself.

Explicitly Specifying the Size and Location of Widgets

The XSIZE (and SCR_XSIZE), YSIZE (and SCR_YSIZE), XOFFSET, and YOFFSET keywords, when used with a standard base widget parent (a base created without the COLUMN or ROW keywords—also called a "bulletin board" base), allow you to specify exactly how the child widgets should be positioned. Sometimes this is a very useful option. However, in general, it is best to avoid this style of programming. Although these keywords are usually honored, they are merely hints to the widget toolkit and might be ignored.

Note

Draw widgets are the exception to this recommendation. In almost all cases, you will want to set the size of draw widgets explicitly, using the sizing keywords.

Explicitly specifying the size and offset makes a program inflexible and unable to run gracefully on various platforms. Often, a layout of this type will look good on one platform, but variations in screen size and how the toolkit works will cause widgets to overlap and not look good on another platform. The best way to handle this situation is to use nested row and column bases to hold the widgets and let the widgets arrange themselves. Such bases are created using the COLUMN and ROW keywords to the WIDGET_BASE function.

Sizing Keywords

When explicitly setting the size of a widget, IDL allows you to control three aspects of the size:

- The *virtual size* is the size of the *potentially* viewable area of the widget. The virtual size may be larger than the actual viewable area on your screen. The virtual size of a widget is determined by either the widget's value, or the XSIZE and YSIZE keywords to the widget creation routine.
- The *viewport size* is the size of the viewable area on your screen. If the viewport size is smaller than the virtual size, scroll bars may be present to allow you to view different sections of the viewable area. When creating widgets for which scroll bars are appropriate, you can add scroll bars by setting the either SCROLL keyword or the APP_SCROLL keyword to the widget creation routine. (For information on the difference, see "Scrolling Draw Widgets" on page 817.) You can explicitly set the size of the viewport area using the X_SCROLL_SIZE and Y_SCROLL_SIZE keywords when creating base, draw, and table widgets.
- The *screen size* is the size of the widget on your screen. You can explicitly specify a screen size using the SCR_XSIZE and SCR_YSIZE keywords to the widget creation routine. Explicitly-set viewport sizes (set with X_SCROLL_SIZE or Y_SCROLL_SIZE) are ignored if you specify the screen size.

The following code shows an example of the WIDGET_DRAW command:

```
draw = WIDGET_DRAW(base, XSIZE=384, YSIZE=384,$
  X_SCROLL_SIZE=192, Y_SCROLL_SIZE = 192, SCR_XSIZE=200)
```

This results in the following:

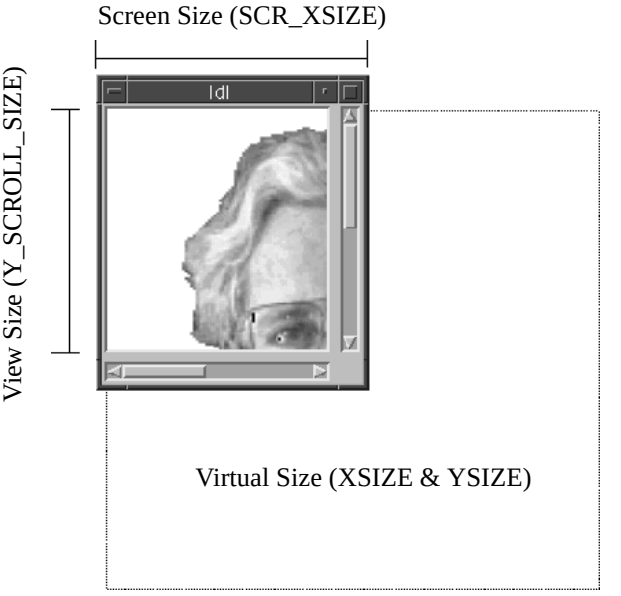


Figure 27-7: Visual description of widget sizes.

In this case, the XSIZE and YSIZE keywords set the virtual size to 384 x 384 pixels. The X_SCROLL_SIZE and Y_SCROLL_SIZE keywords set the viewport size to 192 x 192 pixels. Finally, the SCR_XSIZE keyword overrides the X_SCROLL_SIZE keyword and forces the screen size of the widget (in the X-dimension) to 200 pixels, including the scroll bar.

Controlling Widget Size after Creation

A number of keywords to the WIDGET_CONTROL procedure allow you to change the size of a widget after it has been created. (You will find a list of the keywords to WIDGET_CONTROL that apply to each type of widget at the end of the widget creation routine documentation.) Note that keywords to WIDGET_CONTROL may not control the same parameters as their counterparts associated with widget creation routines. For example, while the XSIZE and YSIZE keywords to WIDGET_DRAW control the virtual size of the draw widget, the XSIZE and YSIZE keywords to WIDGET_CONTROL (when called with the widget ID of a draw widget) control the viewport size of the draw widget. See the *IDL Reference Guide* for details.

Units of Measurement

You can specify the unit of measurement used for most widget sizing operations. When using a widget creation routine, or when using WIDGET_CONTROL or WIDGET_INFO, set the UNITS keyword equal to 0 (zero) to specify that all measurements are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

Note

The UNITS keyword does not affect all sizing operations. Specifically, the value of UNITS is ignored when setting the XSIZE or YSIZE keywords to WIDGET_LIST, WIDGET_TABLE, or WIDGET_TEXT.

Finding the Size of the Screen

When creating the top-level base for an application, sometimes it is useful to know the size of the screen. This information is available via the GET_SCREEN_SIZE function. GET_SCREEN_SIZE returns a two-element floating-point array specifying the size of the screen, in pixels. See "GET_SCREEN_SIZE" in the *IDL Reference Guide* manual for details.

Preventing Layout Flicker

After a widget hierarchy has been realized, adding or destroying widgets in that hierarchy causes IDL to recalculate and set new geometries for every widget in the hierarchy. When a number of widgets are added or destroyed, these calculations occur between each change to the hierarchy, resulting in unpleasant screen "flashing" as the user sees a brief display of each intermediate widget configuration. This behavior can be eliminated by using the UPDATE keyword to WIDGET_CONTROL.

The top-level base of every widget hierarchy has an UPDATE attribute that determines whether or not changes to the hierarchy are displayed on screen. Setting UPDATE to 0 turns off immediate updates and allows you to make a large number of changes to a widget hierarchy without updating the screen after each change. After all of your changes have been made, setting UPDATE to 1 causes the final widget configuration to be displayed on screen.

For example, consider the following main-level program that realizes an unmapped base, then adds 200 button widgets to the previously-realized base:

```
time = SYSTIME(1)
b = WIDGET_BASE(/COLUMN, XSIZE=400, YSIZE=400, MAP=0)
WIDGET_CONTROL, b, /REALIZE
```

```
FOR i = 0, 200 DO button = WIDGET_BUTTON(b, VALUE=STRING(i))
WIDGET_CONTROL, b, /MAP
PRINT, 'time used: ', SYSTIME(1) - time
END
```

This program takes over 50 seconds to run on an HP 9000/720 workstation. If the base had been mapped, the user would see the base “flashing” as each button was added to the base. Altering the example to use the UPDATE keyword reduces the execution time to 0.7 seconds and eliminates the flashing:

```
time = SYSTIME(1)
b = WIDGET_BASE(/COLUMN, XSIZE=400, YSIZE=400, MAP=0)
WIDGET_CONTROL, b, /REALIZE, UPDATE=0
FOR i = 0, 200 DO button = WIDGET_BUTTON(b, VALUE=STRING(i))
WIDGET_CONTROL, b, /MAP, /UPDATE
PRINT, 'time used: ', SYSTIME(1) - time
END
```

Note

Do not attempt to resize a widget on the Windows platform while UPDATE is turned off. Doing so may prevent IDL from updating the screen properly.

Tips on Creating Widget Applications

The following are some ideas to keep in mind when writing widget applications in IDL.

- When writing new applications, decompose the problem into sub-problems and write reusable compound widgets to implement them. In this way, you will build a collection of reusable widget solutions to general problems instead of hard-to-modify, monolithic programs.
- Use the GROUP_LEADER keyword to WIDGET_BASE to define the relationships between parts of your application. Group leadership/membership relationships make it easy to group widgets appropriately for iconization, layering, and destruction.
- Use the MBAR keyword to WIDGET_BASE to create application-specific menubars. Use keyboard focus events to track which widget menu options should affect.
- Use existing compound widgets when possible. In particular, use the CW_BGROUP and CW_PDMENU compound widgets to create menus. These functions are easier to use than writing the menu code directly, and your intent will be more quickly understood by others reading your code.
- The many advantages of the XMANAGER procedure dictate that all widget programs should use it. There are few if any reasons to call the WIDGET_EVENT procedure directly.
- Use CATCH to handle any unanticipated errors. The CATCH branch can free any pointers, pixmaps, logical units, etc., to which the calling routine will not have access, and restore IDL session-wide settings like color tables and system variables that were locally modified.
- It can be difficult to write 100% portable widget code that looks good on all platforms, so let IDL do the layout for you when possible. If all else fails, it is possible to use the value of the WIDGET_INFO function to execute special-case code for each platform's user interface toolkit. It is desirable, however, to avoid large-scale special-case programming because this makes maintenance of the finished program more difficult. See "Portability Issues" below for additional suggestions.

Portability Issues

Although IDL widgets are essentially the same on all supported platforms, there are some differences that can complicate writing applications that work well everywhere. The following hints should help you write such applications:

- Avoid specifying the absolute size and location of widgets whenever possible. (That is, avoid using the XSIZE, YSIZE, XOFFSET, and YOFFSET keywords.) The different user interface toolkits used by different platforms create widgets with slightly different sizes and layouts, so it is best to use bases that order their child widgets in rows or columns and stay away from explicit positioning. If you must use these keywords, try to isolate the affected widgets in a sub-base of the overall widget hierarchy to minimize the overall effect.
- When using a bitmap to specify button labels, be aware that some toolkits prefer certain sizes and give sub-optimal results with others.
- Try to place text, label, and list widgets in locations where their absolute size can vary without making the overall application look bad. The fonts used by the different toolkits have different physical sizes that can cause these widgets to have different proportions.

It is reasonably easy to write applications that will work in all environments without having to resort to much special-case programming. It is very helpful to have a machine running each environment available so that the design can be tested on each iteratively until a suitable layout is obtained.

Using Button Widgets

Button widgets allow users to respond to “yes-or-no” type questions via the widget interface. While button widgets are generally fairly simple to understand and use, there are numerous options that allow you to fine-tune the appearance and behavior of buttons in your interface. This section discusses some useful ideas and techniques for using button widgets. See “WIDGET_BUTTON” in the *IDL Reference Guide* manual for a complete description of the function used to create button widgets.

This section discusses the following topics:

Bitmap Button Labels

In addition to setting the VALUE of a button widget to a text string, you can use a bitmap image as the label for the button. To us a bitmap image, set VALUE to one of the following:

- the path to a bitmap image file, if the BITMAP keyword is also specified.
- an *n* x *m* byte array converted to a bitmap byte array using the CVTTOBM function, which displays as a black-and-white bitmap image.
- an *n* x *m* x 3 byte array, which displays as a 24-bit color bitmap image.

The following sections describe the process of creating bitmap files, black-and-white arrays, and color arrays for use as bitmap button labels.

Creating Bitmap Files for Buttons

You can produce appropriate bitmap files (for use with the BITMAP keyword to WIDGET_BUTTON) using any bitmap editor available on your operating system. Be sure to save the file as a `.bmp` file.

Additionally, on Windows, you can create a bitmap using the IDL GUIBuilder Bitmap Editor, which creates 16-color bitmaps for buttons. The Bitmap Editor can read and write bitmap files (`*.bmp`). Using the editor, you can create your own bitmaps, or you can open existing bitmap files and modify them. Open the Bitmap Editor from the Properties dialog for a created button. For more information, see “Using the Bitmap Editor” in Chapter 24 of the *Building IDL Applications* manual.

Transparent Bitmaps

For 16- and 256-color bitmaps included using the BITMAP keyword, IDL uses the color of the pixel in the lower left corner as the transparent color. All pixels of this color become transparent, allowing the button color to show through. This allows you to use bitmaps that do not appear to be rectangular. If you have a rectangular bitmap that you want to use as a button label, you must either draw a border of a different color around the bitmap or save the bitmap as a 24-bit (TrueColor) image. If your bitmap also contains text, make sure the border you draw is a different color than the text, otherwise the text color will become transparent.

Note

The IDL GUIBuilder's bitmap editor creates 16-color bitmaps.

Creating Black-and-White Bitmap Arrays for Buttons

You can produce appropriate black-and-white bitmap arrays in IDL in the following ways:

- Create a black and white bitmap using an external bitmap editor, and read it into an IDL byte array using the appropriate procedure (READ_BMP, READ_JPEG, etc.) and convert the byte array to a bitmap byte array using the CVTTOBM function.
- On an X-Window system, use the X11 bitmap utility to create a black and white bitmap byte array and read it in to IDL using the READ_X11_BITMAP routine.
- Create a black and white bitmap using the XBM_EDIT procedure. This procedure offers several alternatives for the form of the final bitmap.
- Create an *n* x *m* byte array using the BYTARR function and modify array elements using array operations. Use CVTTOBM to convert the array to a bitmap byte array.

Creating Color Bitmap Arrays for Buttons

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

- Create a 24-bit color image using an external bitmap editor, and read it into an IDL byte array using the appropriate procedure (READ_BMP, READ_JPEG, etc.). Remember that the image array must be interleaved by plane (*n* x *m* 3), with the planes in the order red, green, blue. Note that image files created by image editors are often interleaved by pixel rather than by plane; use the TRANSPOSE function to reformat the array.

For example, if you read a 24-bit color image into an array using the READ_BMP function, the resulting array will be interleaved by pixel (with dimensions 3 x *n* x *m*), with planes in the order blue, green, red. To create an array in the proper format for use as a button bitmap, use the following IDL commands:

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

Here, the RGB keyword to READ_BMP reorders the color planes to be in the order red, green, blue; the call to TRANSPOSE puts the array in the proper format for use in a bitmap button.

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

Although IDL places no restriction on the size of bitmap allowed, the various toolkits may prefer certain sizes.

Tooltips

You can specify a “tooltip” — a short text string that will appear when the mouse pointer hovers over a button widget — by specifying the string as the value of the TOOLTIP keyword to WIDGET_BUTTON.

Figure 27-8: A tool tip.

Note

Tooltips cannot be created for menu sub-items. The topmost button of a pulldown menu can, however, have a tooltip.

Exclusive and Non-Exclusive Buttons

By default, when a user clicks on a button widget, the button appears to be depressed while the user holds down the mouse button, but the button returns to the undepressed appearance when the user releases the mouse button. While such “normal” buttons visually reflect the state of the button (depressed or undepressed), normal buttons are used to gather a single piece of information: whether the user clicked on the button or not.

Buttons placed into exclusive or non-exclusive bases (created via the EXCLUSIVE or NONEXCLUSIVE keywords to WIDGET_BASE procedure) are created as two-state “toggle” buttons. Visually, when a user clicks on an exclusive or nonexclusive button, it remains in the depressed state, either until the user clicks on it again or (in the case of exclusive buttons) until another button in the group is depressed. Buttons that toggle in this manner can be used to gather information about a quantity that has two possible states.

Exclusive and nonexclusive buttons differ in the following way:

- If a base is created with the EXCLUSIVE keyword, only one button on the base can be selected at a given time. If one button is selected and another button pressed, the first button becomes unselected.
- If a base is created with the NONEXCLUSIVE keyword, any number of buttons can be selected at a given time. Pressing one button has no effect on the selected/unselected state of other buttons on the base.

Exclusive and nonexclusive buttons take on different appearances, depending on the type of button and on the windowing toolkit in use (Microsoft Windows or Motif).

Often, it is easier to create groups of buttons (normal, exclusive, or nonexclusive) using the CW_BGROUP compound widget than it is to program them yourself from base and button widgets and manage the events from each button individually. See “Button Groups” on page 790 and “CW_BGROUP” in the *IDL Reference Guide* manual for additional information on using button groups.

Using Draw Widgets

Draw widgets are graphics windows that appear as part of a widget hierarchy rather than appearing as an independent window. Like other graphics windows, draw widgets can be created to use either Direct or Object graphics. (See Chapter 16, “Graphics” in the *Using IDL* manual for a discussion of IDL’s two graphics modes.) Draw widgets allow designers of IDL graphical user interfaces to take advantage of the full power of IDL graphics in their displays. See “WIDGET_DRAW” in the *IDL Reference Guide* manual for a complete description of the function used to create draw widgets.

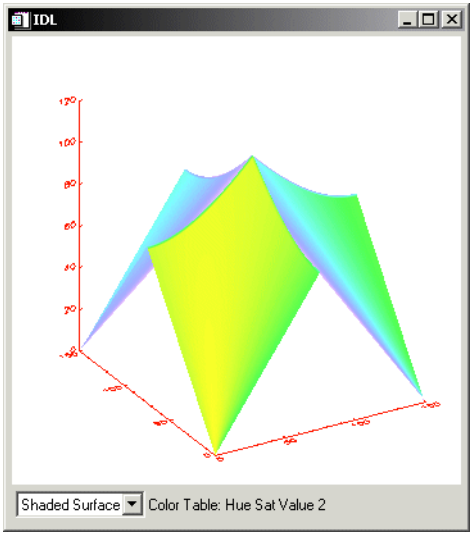


Figure 27-9: An IDL draw widget displaying a shaded surface.

This section discusses the following topics:

Using Direct Graphics in Draw Widgets

By default, draw widgets use IDL Direct graphics. (To create a draw widget that uses Object graphics, set the GRAPHICS_LEVEL keyword to WIDGET_DRAW equal to two; see "Using Object Graphics in Draw Widgets" on page 816.) Once created, draw widgets using Direct graphics are used in the same way as standard Direct graphics windows created using the WINDOW procedure.

All IDL Direct graphics windows are referred to by a window number. Unlike windows created by the WINDOW procedure, the window number of a Direct graphics draw widget cannot be assigned by the user. In addition, the window number of a draw widget is not assigned until the draw widget is actually realized, and thus cannot be returned by WIDGET_DRAW when the widget is created. Instead, you must use the WIDGET_CONTROL procedure to retrieve the window number, which is stored in the *value* of the draw widget, *after* the widget has been realized.

Unlike normal graphics windows, creating a draw widget does not cause the current graphics window to change to the new widget. You must use the WSET procedure to explicitly make the draw widget the current graphics window. The following IDL statements demonstrate the required steps:

```
;Create a base widget.
base = WIDGET_BASE()

;Attach a 256 x 256 draw widget.
draw = WIDGET_DRAW(base, XSIZE = 256, YSIZE = 256)

;Realize the widgets.
WIDGET_CONTROL, /REALIZE, base

;Obtain the window index.
WIDGET_CONTROL, draw, GET_VALUE = index

;Set the new widget to be the current graphics window
WSET, index
```

If you attempt to get the value of a draw widget before the widget has been realized, WIDGET_CONTROL returns the value -1, which is not a valid index.

Using Object Graphics in Draw Widgets

To create a draw widget that uses Object graphics, set the GRAPHICS_LEVEL keyword to WIDGET_DRAW equal to two. Once created, draw widgets using Object graphics are used in the same way as standard IDLgrWindow objects.

All IDL Object graphics windows (that is, IDLgrWindow objects) are referred to by an object reference. Since you do not explicitly create the IDLgrWindow object used in a draw widget, you must retrieve the object reference by using the WIDGET_CONTROL procedure to get the *value* of the draw widget. As with Direct graphics draw widgets, the window object is not created—and thus the object reference cannot be retrieved—until after the draw widget is realized. If you attempt to retrieve the object reference for a draw widget's IDLgrWindow object before the draw widget is realized, IDL returns a null object.

Scrolling Draw Widgets

Another difference between a draw widget and either a graphics window created with the WINDOW procedure or an IDLgrWindow object is that draw widgets can include scroll bars. Setting the APP_SCROLL keyword or the SCROLL keyword to the WIDGET_DRAW function causes scrollbars to be attached to the drawing widget, which allows the user to view images or graphics larger than the visible area.

Differences Between SCROLL and APP_SCROLL

The amount of memory used by a draw widget is directly related to the size of the drawable area of the widget. If a draw widget does not have scroll bars, the entire drawable area is viewable. In this case, the size of the drawable area is controlled by the XSIZE and YSIZE keywords to WIDGET_DRAW.

With the addition of scroll bars, it is possible to display an image that is larger than the viewable area (the *viewport*) of the draw widget. IDL provides two options for dealing with images larger than the viewport:

1. Create the draw widget using the SCROLL keyword. This method creates a draw widget whose drawable area is specified by the XSIZE and YSIZE keywords, and whose viewable area is specified by the X_SCROLL_SIZE and Y_SCROLL_SIZE keywords. Since the entire image is kept in memory, IDL can display the appropriate portions automatically when the scroll bars are adjusted.
2. Create the draw widget using the APP_SCROLL keyword. This method creates a draw widget whose drawable area is the same size as its viewable area (specified by the X_SCROLL_SIZE and Y_SCROLL_SIZE keywords), but which can be different from the *virtual drawable area* (specified by the XSIZE and YSIZE keywords) that is equal to the full size of the image. In this case, only the portion of the image that is currently visible in the viewport is kept in memory; the IDL programmer must use viewport events to determine when the

scroll bars have been adjusted and display the appropriate portion of the full image.

The concept of a virtual drawable area allows you to display portions of very large images in a draw widget without the need for enough memory to display the entire image. The price for this facility is the need to manually handle display of the correct portion of the image in an event-handling routine.

Example using SCROLL

The following code creates a simple scrollable draw widget and displays an image.

Note

This example is included in the file `draw_scroll.pro` in the `examples/widgets` subdirectory of the IDL distribution. You can either open the file in an IDL editor window and compile and run the code using items on the **Run** menu, or simply enter

```
draw_scroll
```

at the IDL command prompt. See "Running the Example Code" on page 738 if IDL does not run the program as expected. You may need to enter `DEVICE, RETAIN=2` at the IDL command prompt before running this example.

```
; Event-handler routine. Does nothing in this example.
PRO draw_scroll_event, ev

END

; Widget creation routine.
PRO draw_scroll

  ; Read an image for use in the example.
  READ_JPEG, FILEPATH('muscle.jpg', $
    SUBDIR=['examples', 'data']), image

  ; Create the base widget.
  base = WIDGET_BASE()

  ; Create the draw widget. The size of the viewport is set to
  ; 200x200 pixels, but the size of the drawable area is
  ; set equal to the dimensions of the image array using the
  ; XSIZE and YSIZE keywords.
  draw = WIDGET_DRAW(base, X_SCROLL_SIZE=200, Y_SCROLL_SIZE=200, $
    XSIZE=(SIZE(image))[1], YSIZE=(SIZE(image))[2], /SCROLL)

  ; Realize the widgets.
  WIDGET_CONTROL, base, /REALIZE
```

```
    ; Retrieve the window ID from the draw widget.
    WIDGET_CONTROL, draw, GET_VALUE=drawID

    ; Set the draw widget as the current drawable area.
    WSET, drawID

    ; Load the image.
    TVSCL, image

    ; Call XMANAGER to manage the widgets.
    XMANAGER, 'draw_scroll', base, /NO_BLOCK

END
```

In this example, the drawable area created for the draw widget is the full size of the displayed image. Since IDL handles the display of the image as the scroll bars are adjusted, no event-handling is necessary to update the display.

Example using APP_SCROLL

We can easily rework the previous example to use the APP_SCROLL keyword rather than the SCROLL keyword. Using APP_SCROLL has the following consequences:

1. IDL no longer automatically displays the appropriate portion of the image when the scroll bars are adjusted. As a result, we must add code to our event-handling procedure to check for the viewport event and display the appropriate part of the image. Here is the new event-handler routine:

   ```
   ; Event-handler routine.
   PRO draw_app_scroll_event, ev

     COMMON app_scr_ex, image

     IF (ev.TYPE EQ 3) THEN TVSCL, image, 0-ev.X, 0-ev.Y

   END
   ```

   First, notice that since we need access to the image array in both the widget creation routine and the event handler, we place the array in a COMMON block. This is appropriate since the image data itself is not altered by the widget application.

   Second, we check the `TYPE` field of the event structure to see if it is equal to `3`, which is the code for a viewport event. If it is, we use the values of the `X` and `Y` fields of the event structure as the Position arguments to the TVSCL routine to display the appropriate portion of the image array.

2. We must add the COMMON block to the widget creation routine.

3. We change the call to WIDGET_DRAW to include the APP_SCROLL keyword rather than the SCROLL keyword. In this context, the values of the XSIZE and YSIZE keywords are interpreted as the size of the *virtual* drawable area, rather than the actual drawable area.

Note

The modified example is included in the file `draw_app_scroll.pro` in the `examples/widgets` subdirectory of the IDL distribution.

On the surface the two examples appear identical. The difference is that the example using APP_SCROLL uses only the memory necessary to create the smaller drawable area described by the size of the viewport, whereas the example using SCROLL uses the memory necessary to create the full drawable area described by the XSIZE and YSIZE keywords. While the example image is not so large that this makes much difference, if the image contained several hundred million pixels rather than a few hundred thousand, the memory saving could be significant.

Context Events in Draw Widgets

The WIDGET_DRAW function does not have a CONTEXT_EVENTS keyword to specify that context menu events be generated when the user clicks the right mouse button over a drawable area. Instead, the event structure generated by draw widgets when the BUTTON_EVENTS keyword is set includes the PRESS and RELEASE fields, both of which contain information regarding which mouse button was pressed.

See “Context-Sensitive Menus” on page 795 for techniques used to simulate the generation of context menu events with draw widgets.

Draw Widget Example

The following example program creates a small widget application consisting of a draw widget and a droplist menu. One of three plots is displayed in the draw widget depending on the selection made from the droplist. To add to dynamic behavior, we will use timer events to change the color table used in the draw window every three seconds.

Note

This example is included in the file `draw_widget_example.pro` in the `examples/widgets` subdirectory of the IDL distribution. You can either open the file in an IDL editor window and compile and run the code using items on the **Run** menu, or simply enter

```
draw_widget_example
```

at the IDL command prompt. See “Running the Example Code” on page 738 if IDL does not run the program as expected.

```
; Event-handler routine
PRO draw_widget_example_event, ev

  ; We need to save the value of the seed variable for the random
  ; number generator between calls to the event-handling routine.
  ; We do this using a COMMON block.

  COMMON dwe, seed

  ; Retrieve the anonymous structure contained in the user value of
  ; the top-level base widget. This structure contains the
  ; following fields:
  ;    drawID:   the widget ID of the draw widget
  ;   labelID:   the widget ID of the label widget that will hold
  ;              the color table name.
  ;   sel_index: the index of the current selection in the
  ;              droplist
  ;   ctable:    the index of the current color table.

  WIDGET_CONTROL, ev.TOP, GET_UVALUE=stash

  ; Set the draw widget as the current IDL drawable.

  WSET, stash.drawID

  ; Check the type of event structure returned. If it is a timer
  ; event, change the color table index to a random number between
  ; 0 and 40, then set the value of the label widget to the name of
  ; the new color table.
  ; (See “Identifying Widget Type from an Event” on page 783 for
  ; more on identifying widget types from returned event
  ; structures.)

  IF (TAG_NAMES(ev, /STRUCTURE_NAME) EQ 'WIDGET_TIMER') THEN BEGIN
    LOADCT, GET_NAMES=ctnames
    stash.ctable = FIX(RANDOMU(seed)*41)
    LOADCT, stash.ctable, /SILENT
    WIDGET_CONTROL, stash.labelID, $
      SET_VALUE='Color Table: ' + ctnames[stash.ctable]
    WIDGET_CONTROL, ev.ID, TIMER=3.0
  ENDIF

  ; If the event is a droplist event, change the value of the
  ; variable 'selection' to the new index value.
```

```
  IF (TAG_NAMES(ev, /STRUCTURE_NAME) EQ 'WIDGET_DROPLIST') $
    THEN BEGIN
    stash.sel_index=ev.index
  ENDIF

  ; Reset the user value of the top-level base widget to the
  ; modified stash structure.

  WIDGET_CONTROL, ev.TOP, SET_UVALUE=stash

  ; Display a plot, surface, or shaded surface, or destroy the
  ; widget application, depending on the value of the 'selection'
  ; variable.

  CASE stash.sel_index OF
    0: PLOT, DIST(150)
    1: SURFACE, DIST(150)
    2: SHADE_SURF, DIST(150)
    3: WIDGET_CONTROL, ev.TOP, /DESTROY
  ENDCASE

END

PRO draw_widget_example

  ; Define the values for the droplist widget and define the
  ; initially selected index to show a shaded surface.
  select = ['Plot', 'Surface', 'Shaded Surface', 'Done']
  sel_index = 2

  ; Create a base widget containing a draw widget and a sub-base
  ; containing a droplist menu and a label widget.
  base = WIDGET_BASE(/COLUMN)
  draw = WIDGET_DRAW(base, XSIZE=350, YSIZE=350)
  base2 = WIDGET_BASE(base, /ROW)
  dlist = WIDGET_DROPLIST(base2, VALUE=select)
  label = WIDGET_LABEL(base2, XSIZE=200)

  ; Realize the widget hierarchy, then retrieve the widget ID of
  ; the draw widget.
  WIDGET_CONTROL, base, /REALIZE
  WIDGET_CONTROL, draw, GET_VALUE=drawID

  ; Set the timer value of the draw widget.
  WIDGET_CONTROL, draw, TIMER=0.0

  ; Set the droplist to display the proper selection index.
  WIDGET_CONTROL, dlist, SET_DROPLIST_SELECT=sel_index
```

```
    ; Store the widget ID of the draw widget, the widget ID of
    ; the label widget, the droplist selection index, and the
    ; initial color table index in an anonymous structure, and
    ; set the user value of the top-level base widget to this
    ; structure.
    stash = { drawID:drawID, labelID:label, $
              sel_index:sel_index, ctable:0}
    WIDGET_CONTROL, base, SET_UVALUE=stash

    ; Register the widget with the XMANAGER.
    XMANAGER, 'draw_widget_example', base, /NO_BLOCK

    ; Set some display device parameters.
    DEVICE, RETAIN=2, DECOMPOSED=0

END
```

The intent of this example is to demonstrate the use of draw widgets, menus, and timer events with a minimum of other complicating issues. However, it is easy to imagine applications wherein a graphics window containing a plot or some other information is updated periodically by a timer. The method used here can be easily applied to more realistic situations.

Button, Motion, and Keyboard Events

To go beyond merely displaying an image in a draw widget and allow the user to interact in some way with the displayed image, you must configure the draw widget to generate either *button*, *motion*, or *keyboard* events:

- *Button events* are enabled by setting the BUTTON_EVENTS keyword to WIDGET_DRAW. Once enabled, button events are generated when the user clicks on the draw widget.
- *Motion events* are enabled by setting the MOTION_EVENTS keyword to WIDGET_DRAW. Once enabled, motion events are generated whenever the cursor moves over the draw widget.
- *Keyboard events* are enabled by setting the KEYBOARD_EVENTS keyword to WIDGET_DRAW. Once enabled, events are generated when the draw widget has focus and a keyboard key is pressed.

The following example uses motion events to update the values of several label widgets as the mouse cursor moves over an image in a draw widget. This and several other features are discussed in the section following the code.

Note

This example is included in the file `draw_widget_data.pro` in the `examples/widgets` subdirectory of the IDL distribution. You can either open the file in an IDL editor window and compile and run the code using items on the **Run** menu, or simply enter

```
draw_widget_data
```

at the IDL command prompt. See "Running the Example Code" on page 738 if IDL does not run the program as expected. You may need to enter `DEVICE, DECOMPOSED=1` at the IDL command prompt before running this example.

```
; Event-handling procedure.
PRO draw_widget_data_event, ev

  ; Retrieve the anonymous structure contained in the user value of
  ; the top-level base widget.

  WIDGET_CONTROL, ev.TOP, GET_UVALUE=stash

  ; If the event is generated in the draw widget, update the
  ; label values with the current cursor position and the value
  ; of the data point under the cursor. Note that since we have
  ; passed a pointer to the image array rather than the array
  ; itself, we must dereference the pointer in the 'image' field
  ; of the stash structure before getting the subscripted value.

  IF (TAG_NAMES(ev, /STRUCTURE_NAME) eq 'WIDGET_DRAW') THEN BEGIN
    WIDGET_CONTROL, stash.label1, $
      SET_VALUE='X position: ' + STRING(ev.X)
    WIDGET_CONTROL, stash.label2, $
      SET_VALUE='Y position: ' + STRING(ev.Y)
    WIDGET_CONTROL, stash.label3, $
      SET_VALUE='Hex Value: ' + $
      STRING((*stash.imagePtr)[ev.X, ev.Y], FORMAT='(Z12)')
  ENDIF

  ; If the event is generated in a button, destroy the widget
  ; hierarchy. We know we can use this simple test because there
  ; is only one button in the application.

  IF (TAG_NAMES(ev, /STRUCTURE_NAME) eq 'WIDGET_BUTTON') THEN BEGIN
    WIDGET_CONTROL, ev.TOP, /DESTROY
  ENDIF

END

; Widget creation routine.
PRO draw_widget_data
```

```
; Define a monochrome image array for use in the application.
READ_PNG, FILEPATH('mineral.png', $
  SUBDIR=['examples', 'data']), image

; Place the image array in a pointer heap variable, so we can
; pass the pointer to the event routine rather than passing the
; entire image array.
imagePtr=PTR_NEW(image, /NO_COPY)

; Retrieve the size information from the image array.
im_size=SIZE(*imagePtr)

; Create a base widget to hold the application.
base = WIDGET_BASE(/COLUMN)

; Create a draw widget based on the size of the image, and
; set the MOTION_EVENTS keyword so that events are generated
; as the cursor moves across the image. Setting the BUTTON_EVENTS
; keyword rather than MOTION_EVENTS would require the user to
; click on the image before an event is generated.
draw = WIDGET_DRAW(base, XSIZE=im_size[1], YSIZE=im_size[2], $
  /MOTION_EVENTS)

; Create 'Done' button.
button = WIDGET_BUTTON(base, VALUE='Done')

; Create label widgets to hold the cursor position and
; Hexadecimal value of the pixel under the cursor.
label1 = WIDGET_LABEL(base, XSIZE=im_size[1]*.9, $
  VALUE='X position:')
label2 = WIDGET_LABEL(base, XSIZE=im_size[1]*.9, $
  VALUE='Y position:')
label3 = WIDGET_LABEL(base, XSIZE=im_size[1]*.9, $
  VALUE='Hex Value:')

; Realize the widget hierarchy.
WIDGET_CONTROL, base, /REALIZE

; Retrieve the widget ID of the draw widget. Note that the widget
; hierarchy must be realized before you can retrieve this value.
WIDGET_CONTROL, draw, GET_VALUE=drawID

; Create an anonymous array to hold the image data and widget IDs
; of the label widgets.
stash = { imagePtr:imagePtr, label1:label1, label2:label2, $
          label3:label3 }

; Set the user value of the top-level base widget equal to the
```

```
    ; 'stash' array.
    WIDGET_CONTROL, base, SET_UVALUE=stash

    ; Make the draw widget the current IDL drawable area.
    WSET, drawID

    ; Draw the image into the draw widget.
    TVSCL, *imagePtr

    ; Call XMANAGER to manage the widgets.
    XMANAGER, 'draw_widget_data', base, /NO_BLOCK

END
```

The following things about this example are worth noting:

- Since we use the image data in both the widget creation routine (where we display the image) and the event-handler routine (where we retrieve the value of the data point under the cursor), we need access to the variable that holds the image in both places. We could pass the entire image array from the creation routine to the event-handler in the `stash` structure, but since the image could be large, we choose to pass a *pointer* to the image instead. This means we must dereference the pointer variable every time we need to use the image data. For more information on pointers and how to dereference them, see Chapter 8, “Pointers”.
- In this example we have set the MOTION_EVENTS keyword to WIDGET_DRAW; this causes events to be generated continuously as the cursor moves across the draw widget. We could have set the BUTTON_EVENTS keyword instead; this would force the user to click the draw widget in order to update the text fields.

Using Property Sheet Widgets

The purpose of a property sheet is to enable the user to view and edit the properties of an object subclassed from the IDLitComponent class. (All IDLgr* and IDLit* objects subclass from the IDLitComponent class.)

For example, a user may have rendered data as a surface. Using IDL's iSurface tool, the user can select the surface and bring up a property sheet that lists all of the surface's properties, including color, shading method, etc. To change the color, the user can go to the property sheet, select the color property, bring up the color picker, and select a new color. The name of the changed property is placed into an IDL event. It is in the processing of this event that the object is updated. An existing property sheet can be assigned a new component, which causes it to reload with the new list of properties and their values.

The following topics show how to use the property sheet widget with the iTool's paradigm:

Registering Properties

In order for a property associated with a component object to be included in the property sheet for that component, the property must be *registered*. The property registration mechanism accomplishes several things:

- It allows you to expose as many or as few of the properties of an underlying object as you choose.
- It allows you to add user-defined properties to existing objects, and expose those new properties to users of your application.

Groups of properties of graphical atomic objects can be registered by setting their REGISTER_PROPERTIES properties to True when the object is initialized. See the property tables for each graphical atomic object in the *IDL Reference Guide*.

Selecting Properties

A select event is generated whenever the property sheet's current row or column changes. Navigation between cells is performed by using the mouse to left click on a cell. When the property sheet is initially realized, no selected cell exists.

The event structure (WIDGET_PROPSHEET_SELECT) provided when selection occurs contains a COMPONENT and an IDENTIFIER tag. The COMPONENT tag is a reference to the object associated with the property sheet. When multiple objects are associated with the property sheet, this member indicates which one object had one of its properties selected. The IDENTIFIER tag uniquely identifies the property. This identifier is unique among all of the component's properties. The component and identifier can be used to obtain the value of the selected property:

```
isDefined = event.component -> $
   GetPropertyByIdentifier(event.identifier, value)
```

where `event` is the event structure, `isDefined` is a 1 if the value is defined (0, otherwise), and `value` receives the property's value.

Changing Properties

A change event is generated whenever a new value is entered for a property. It is also used to signal that a user-defined property needs changing.

The event structure (WIDGET_PROPSHEET_CHANGE) provided when a change occurs contains a COMPONENT, an IDENTIFIER, a PROPTYPE, and a SET_DEFINED tag. The COMPONENT tag contains a reference to the object associated with the property sheet. When multiple objects are associated with the property sheet, this member indicates which object is to change. The IDENTIFIER tag specifies the value of the property's identifier attribute. This identifier is unique among all of the component's properties. The PROPTYPE tag indicates the type of the property (integer, string, etc.). Integer values for these types can be found in the documentation for components. The SET_DEFINED tag indicates whether or not an undefined property is having its value set. In most circumstances, along with its new value, the property should have its 'UNDEFINED' attribute set to zero. If a property is never marked as undefined, this field can be ignored.

Although the component's object reference is included in the event structure, it can also be retrieved via the following call:

```
WIDGET_CONTROL, event.id, GET_VALUE = obj
```

where `event` is the event structure and `obj` is the object reference of the component.

The PROPTYPE field is provided for convenience. The property type should be known implicitly based on `IDENTIFIER`, but can be retrieved (in integer form) by:

```
obj -> GetPropertyAttribute, event.identifier, TYPE = type
```

where `obj` is the object reference of the component, `event` is the event structure, and `type` represents the data type of the property. Here, the value returned in by the TYPE keyword is the same as the value of the PROPTYPE field of the widget event structure.

Properties can use their UNDEFINED attribute to show an indeterminate state (set attribute UNDEFINED = 1). This might arise after the aggregation of two or more properties. One could imagine a COLOR property representing both the border and the interior color of a polygon so that just one color property is displayed in the property sheet. When set, the chosen color would be applied to both, and then the following code could be used to mark the property as defined:

```
IF (event.set_defined) THEN $
   event.component -> SetPropertyAttribute, $
      event.identifier, UNDEFINED = 0

WIDGET_CONTROL, event.id, REFRESH_PROPERTY = event.identifier
```

where `event` is the event structure.

Note

The REFRESH_PROPERTY keyword to WIDGET_CONTROL is used to refresh the property sheet. This is necessary because although the property sheet knows about its component, it does not directly change the component itself. Just as with changing properties values, the property sheet and underlying component have a clear boundary and can only affect each other through IDL statements.

Properties can also be hidden (removing them from the property sheet entirely) or desensitized (displaying the property in the property sheet, but not allowing the user to change its value). See “Property Attributes” in Chapter 4 of the *iTool Developer’s Guide* manual for additional details.

Updating the Component

When a value has been changed in the property sheet, you can access this resulting value through the WIDGET_INFO function:

```
value = WIDGET_INFO(event.id, PROPERTY_VALUE = event.identifier)
```

where `event` is the event structure. This value can then be used to update the changed property in the component object by calling its SetPropertyByIdentifier method:

```
event.component -> SetPropertyByIdentifier, event.identifier, $
   value
```

where `event` is the event structure and `value` is the modified property value.

User-defined Properties

User-defined properties allow IDL programmers to provide their own custom means for editing a property. One significant difference from other types of properties is that user-defined properties must have a string version of their value. This string value is stored in the USERDEF attribute of the property and must be explicitly updated. The string value is the value displayed in the property sheet. See Chapter 4, “Property Management” in the *iTool Developer’s Guide* manual for further discussion of user-defined properties.

Updating User-defined Properties

Like other property types, user-defined properties generate IDL property sheet change events. The difference is that the IDL event handler cannot query the property sheet for the new value. It must use some other means to determine a new value. Typically this is done through widget code, in which the user is asked to set a value, but virtually any other technique is valid.

When handling change events, determine the property's type using the PROPTYPE field of the widget event structure. Once a value has been acquired, update the component using its SetProperty method. In addition, the string version of the user-defined property’s value should be updated. This is done by executing a statement similar to the following example:

```
eventBase.component -> SetPropertyAttribute, $
   eventBase.identifier, USERDEF = userDefValue
```

where `eventBase` is the event structure of the top-level-base and `userDefValue` is the string representing the user-defined value when the property sheet is refreshed.

Once the underlying component has been updated, the property sheet is ready to be refreshed. Execute a call to update a given property with the current value:

```
WIDGET_CONTROL, propsheet, REFRESH_PROPERTY = eventBase.identifier
```

where `propsheet` is the widget ID of the property sheet widget and `eventBase` is the event structure of the top-level-base.

Property Sheet Example

The following example provides a property sheet containing all the available controls, including user-defined properties of a custom component.

Enter the following text into the IDL Editor:

```
; Property Sheet Demo
;
; This program contains these sections of code:
;
; (1) Definition of the IDLitTester class.
; (2) Methods for handling the user-defined data type.
; (3) Event handlers and main widget program.

;=============================================
; (1) Definition of the IDLitTester class.
;---------------------------------------------
; IDLitTester
;
; Superclasses:
;   IDLitComponent
;
; Subclasses:
;   none
;
; Interfaces:
;   IIDLProperty
;
; Intrinsic Methods:
; none (because it contains no objects)

;---------------------------------------------
; IDLitTester::Init

FUNCTION IDLitTester::Init, _REF_EXTRA = _extra

compile_opt idl2

; Initialize the superclass.
IF (self -> IDLitComponent::Init() ne 1) THEN $
   RETURN, 0

; Create IDLitTester.
; Nothing to do, for now.

; Register properties.
;
; * Only registered properties will show up in the property sheet.
```

```
; * <identifier> must match self.<identifier>.

self -> RegisterProperty, 'BOOLEAN', /BOOLEAN , $
   NAME = 'Boolean', DESCRIPTION = 'TRUE or FALSE'

self -> RegisterProperty, 'COLOR', /COLOR, $
   NAME = 'Color', DESCRIPTION = 'Color (RGB)'

self -> RegisterProperty, 'USERDEF', USERDEF = '', $
   NAME = 'User Defined', DESCRIPTION = 'User defined property'

self -> RegisterProperty, 'NUMBER1', /INTEGER , $
   NAME = 'Integer', DESCRIPTION = 'Integer in [-100, 100]', $
   valid_range = [-100, 100]

self -> RegisterProperty, 'NUMBER2', /FLOAT, $
   NAME = 'Floating Point', DESCRIPTION = 'Number trackbar', $
   valid_range = [-19.0D, 6.0D, 0.33333333333333D]

self -> RegisterProperty, 'NUMBER3', /FLOAT, $
   NAME = 'Floating Point', $
   DESCRIPTION = 'Double in [-1.0, 1.0]', $
   valid_range = [-1.0D, 1.0D]

self -> RegisterProperty, 'LINESTYLE', /LINESTYLE, $
   NAME = 'Line Style', DESCRIPTION = 'Line style'

self -> RegisterProperty, 'LINETHICKNESS', /THICKNESS , $
   NAME = 'Line Thickness', $
   DESCRIPTION = 'Line thickness (pixels)'

self -> RegisterProperty, 'STRINGOLA', /STRING , $
   NAME = 'String', DESCRIPTION = 'Just some text'

self -> RegisterProperty, 'SYMBOL', /SYMBOL , $
   NAME = 'Symbol', DESCRIPTION = 'Symbol of some sort'

self -> RegisterProperty, 'STRINGLIST', $
   NAME = 'String List', DESCRIPTION = 'Enumerated list', $
   enumlist = ['dog', 'cat', 'bat', 'rat', 'nat', $
   'emu', 'owl', 'pig', 'hog', 'ant']

; Set any property values.
self -> SetProperty, _EXTRA = _extra

RETURN, 1
END

;-------------------------------------------------
```

```
; IDLitTester::Cleanup

PRO IDLitTester::Cleanup

compile_opt idl2

self -> IDLitComponent::Cleanup

END

;---------------------------------------------
; IDLitTester::GetProperty
;
; Implemention for IIDLProperty interface

PRO IDLitTester::GetProperty, $
   boolean = boolean, $
   color = color, $
   userdef = userdef, $
   font = font, $
   number1 = number1, $
   number2 = number2, $
   number3 = number3, $
   linestyle = linestyle, $
   linethickness = linethickness, $
   stringola = stringola, $
   stringlist = stringlist, $
   symbol = symbol, $
   _REF_EXTRA = _extra

compile_opt idl2

IF (arg_present(boolean)) THEN boolean = self.boolean
IF (arg_present(color)) THEN color = self.color
IF (arg_present(userdef)) THEN userdef = self.userdef
IF (arg_present(font)) THEN font = self.font
IF (arg_present(number1)) THEN number1 = self.number1
IF (arg_present(number2)) THEN number2 = self.number2
IF (arg_present(number3)) THEN number3 = self.number3
IF (arg_present(linestyle)) THEN linestyle = self.linestyle
IF (arg_present(linethickness)) $
   THEN linethickness = self.linethickness
IF (arg_present(stringola)) THEN stringola = self.stringola
IF (arg_present(stringlist)) THEN stringlist = self.stringlist
IF (arg_present(symbol)) THEN symbol = self.symbol

; Superclass' properties:
IF (n_elements(_extra) gt 0) THEN $
   self->IDLitComponent::GetProperty, _EXTRA = _extra
```

```
END

;---------------------------------------------
; IDLitTester::SetProperty
;
; Implementation for IIDLProperty interface

PRO IDLitTester::SetProperty, $
   boolean = boolean, $
   color = color, $
   userdef = userdef, $
   font = font, $
   number1 = number1, $
   number2 = number2, $
   number3 = number3, $
   linestyle = linestyle, $
   linethickness = linethickness, $
   stringola = stringola, $
   stringlist = stringlist, $
   symbol = symbol, $
   _REF_EXTRA = _extra

compile_opt idl2

IF (n_elements(boolean) ne 0) THEN self.boolean = boolean
IF (n_elements(color) ne 0) THEN self.color = color
IF (n_elements(userdef) ne 0) THEN self.userdef = userdef
IF (n_elements(font) ne 0) THEN self.font = font
IF (n_elements(number1) ne 0) THEN self.number1 = number1
IF (n_elements(number2) ne 0) THEN self.number2 = number2
IF (n_elements(number3) ne 0) THEN self.number3 = number3
IF (n_elements(linestyle) ne 0) THEN self.linestyle = linestyle
IF (n_elements(linethickness) ne 0) THEN $
   self.linethickness = linethickness
IF (n_elements(stringola) ne 0) THEN self.stringola = stringola
IF (n_elements(stringlist) ne 0) THEN self.stringlist = stringlist
IF (n_elements(symbol) ne 0) THEN self.symbol = symbol

self -> IDLitComponent::SetProperty, _EXTRA = _extra

END

;---------------------------------------------
; IDLitTester__Define

PRO IDLitTester__Define

compile_opt idl2, hidden
```

```
struct = {$
   IDLitTester, $
   inherits IDLitComponent, $
   boolean:0L, $
   color:[0B,0B,0B], $
   userdef:"", $
   number1:0L, $
   number2:0D, $
   number3:0D, $
   linestyle:0L, $
   linethickness:0L, $
   stringola:"", $
   stringlist:0L, $
   symbol:0L $
   }

END

;================================================
; (2) Methods for handling the user-defined data type.
;------------------------------------------------
; UserDefEvent
;
; This procedure is just part of the widget code for
; the user defined property.

PRO UserDefEvent, e

IF (tag_names(e, /structure_name) eq 'WIDGET_BUTTON') $
   THEN BEGIN

   widget_control, e.top, get_uvalue = uvalue
   widget_control, e.id, get_uvalue = numb_ness

   propsheet  = uvalue.propsheet
   component  = uvalue.component
   identifier = uvalue.identifier

   ; Set the human readable value.
   component -> SetPropertyAttribute, $
      identifier, userdef = numb_ness

   ; Set the real value of the component.
   component -> SetPropertyByIdentifier, identifier, numb_ness

   WIDGET_CONTROL, propsheet, refresh_property = identifier
   PRINT, 'Changed: ', uvalue.identifier, ': ', numb_ness
   WIDGET_CONTROL, e.top, /destroy
```

```
ENDIF

END

;-----------------------------------------------
; GetUserDefValue
;
; Creates widgets used to modify the user defined property's
; value.  The value is actually set in UserDefEvent.

PRO GetUserDefValue, e

base = WIDGET_BASE(/row, title = 'Pick a Number', $
   /modal, group_leader = e.top)

one = WIDGET_BUTTON(base, value = 'one', uvalue = 'oneness')
two = WIDGET_BUTTON(base, value = 'two', uvalue = 'twoness')
six = WIDGET_BUTTON(base, value = 'six', uvalue = 'sixness')
ten = WIDGET_BUTTON(base, value = 'ten', uvalue = 'tenness')

; We will need this info when we set the value
WIDGET_CONTROL, base, $
   SET_UVALUE = {propsheet:e.id, $
   component:e.component, $
   identifier:e.identifier}

WIDGET_CONTROL, base, /REALIZE

XMANAGER, 'UserDefEvent', base, event_handler = 'UserDefEvent'

END

;===============================================
; (3) Event handlers and main widget program.
;-----------------------------------------------
;
; Event handling code for the main widget program and
; the main widget program.

;-----------------------------------------------
; prop_event
;
; The property sheet generates an event whenever the user changes
; a value.  The event holds the property's identifier and type, and
; an object reference to the component.
;
; Note: widget_control, e.id, get_value = objref also retrieves an
; object reference to the component.
```

```
PRO prop_event, e

IF (e.type eq 0) THEN BEGIN   ; Value changed

; Get the value of the property identified by e.identifier.

   IF (e.proptype ne 0) THEN BEGIN

      ; Get the value from the property sheet.
      value = widget_info(e.id, property_value = e.identifier)

      ; Set the component's property's value.
      e.component -> SetPropertyByIdentifier, e.identifier, $
         value

      ; Print the change in the component's property value.
      PRINT, 'Changed', e.identifier, ': ', value
   ENDIF ELSE BEGIN

      ; Use alternative means to get the value.
      GetUserDefValue, e

   ENDELSE

ENDIF ELSE BEGIN               ; selection changed

   PRINT, 'Selected: ' + e.identifier
   r = e.component -> GetPropertyByIdentifier(e.identifier, value)
   PRINT, ' Current Value: ', value

ENDELSE

END

;-------------------------------------------------
; refresh_event

PRO refresh_event, e

WIDGET_CONTROL, e.id, get_uvalue = uvalue

uvalue.o -> SetProperty, boolean = 0L
uvalue.o -> SetProperty, color = [255, 0, 46]
uvalue.o -> SetPropertyAttribute, 'userdef', userdef = "Yeehaw!"
uvalue.o -> SetProperty, number1 = 99L
uvalue.o -> SetProperty, number2 = -13.1
uvalue.o -> SetProperty, number3 = 6.5
uvalue.o -> SetProperty, linestyle = 6L
```

```
uvalue.o -> SetProperty, stringola = 'It worked!'
uvalue.o -> SetProperty, stringlist = 6L
uvalue.o -> SetProperty, symbol = 6L

uvalue.o -> SetPropertyAttribute, 'Number1', sensitive = 1
uvalue.o -> SetPropertyAttribute, 'Number2', sensitive = 1

WIDGET_CONTROL, uvalue.prop, $
   REFRESH_PROPERTY = ['boolean', 'color', 'userdef', $
   'number1', 'number2', 'number3', 'linestyle', $
   'stringola', 'stringlist', 'symbol']

END

;-----------------------------------------------
; reload_event

PRO reload_event, e

WIDGET_CONTROL, e.id, GET_UVALUE = uvalue

LoadValues, uvalue.o

WIDGET_CONTROL, uvalue.prop, SET_VALUE = uvalue.o

update_state, e.top, 1

END

;-----------------------------------------------
; hide_event

PRO hide_event, e

WIDGET_CONTROL, e.id, get_uvalue = uvalue

uvalue.o -> SetPropertyAttribute, 'color', /HIDE

WIDGET_CONTROL, uvalue.prop, refresh_property = 'color'

END

;-----------------------------------------------
; show_event

PRO show_event, e

WIDGET_CONTROL, e.id, get_uvalue = uvalue
```

```
uvalue.o -> SetPropertyAttribute, 'color', hide = 0

WIDGET_CONTROL, uvalue.prop, REFRESH_PROPERTY = 'color'

END

;-------------------------------------------------
; clear_event

PRO clear_event, e

update_state, e.top, 0

WIDGET_CONTROL, e.id, GET_UVALUE = uvalue

WIDGET_CONTROL, uvalue.prop, SET_VALUE = OBJ_NEW()

END

;-------------------------------------------------
; psdemo_large_event
;
; Handles resize events for the property sheet demo program.

WIDGET_CONTROL, e.id, GET_UVALUE = base
geo_tlb = WIDGET_INFO(e.id, /GEOMETRY)

WIDGET_CONTROL, base.prop, $
   SCR_XSIZE = geo_tlb.xsize - (2*geo_tlb.xpad), $
   SCR_YSIZE = geo_tlb.ysize - (2*geo_tlb.ypad)

END

;-------------------------------------------------
; sensitivity_event
;
; Procedure to test sensitizing and desensitizing

PRO sensitivity_event, e

WIDGET_CONTROL, e.id, GET_UVALUE = uvalue, GET_VALUE = value

IF (value eq 'Desensitize') THEN b = 0 $
ELSE b = 1

uvalue.o -> SetPropertyAttribute, 'Boolean', sensitive = b
uvalue.o -> SetPropertyAttribute, 'Color', sensitive = b
uvalue.o -> SetPropertyAttribute, 'UserDef', sensitive = b
uvalue.o -> SetPropertyAttribute, 'Number1', sensitive = b
```

```
uvalue.o -> SetPropertyAttribute, 'Number2', sensitive = b
uvalue.o -> SetPropertyAttribute, 'Number3', sensitive = b
uvalue.o -> SetPropertyAttribute, 'LineStyle', sensitive = b
uvalue.o -> SetPropertyAttribute, 'LineThickness', sensitive = b
uvalue.o -> SetPropertyAttribute, 'Stringola', sensitive = b
uvalue.o -> SetPropertyAttribute, 'Symbol', sensitive = b
uvalue.o -> SetPropertyAttribute, 'StringList', sensitive = b

WIDGET_CONTROL, uvalue.prop, $
   refresh_property = ['Boolean', 'Color', 'UserDef', $
   'Number1', 'Number2', 'Number3', 'LineStyle', $
   'LineThickness', 'Stringola', 'Symbol', 'StringList']

END

;-----------------------------------------------
; LoadValues

PRO LoadValues, o

o -> SetProperty, boolean = 1L             ; 0 or 1
o -> SetProperty, color = [200, 100, 50]   ; RGB
o -> SetPropertyAttribute, 'userdef', userdef = ""
; to be set later
o -> SetProperty, number1 = 42L            ; integer
o -> SetProperty, number2 = 0.0            ; double
o -> SetProperty, number3 = 0.1            ; double
o -> SetProperty, linestyle = 4L           ; 5th item (zero based)
o -> SetProperty, linethickness = 4L       ; pixels
o -> SetProperty, stringola = "This is a silly string."
o -> SetProperty, stringlist = 3L          ; 4th item in list
o -> SetProperty, symbol = 4L              ; 5th symbol in list

END

;-----------------------------------------------
; quit_event

PRO quit_event, e

WIDGET_CONTROL, e.top, /DESTROY

END

;-----------------------------------------------
; update_state

PRO update_state, top, sensitive
```

```
WIDGET_CONTROL, top, GET_UVALUE = uvalue

FOR i = 0, n_elements(uvalue.b) - 1 do $
   WIDGET_CONTROL, uvalue.b[i], sensitive = sensitive

END

;-----------------------------------------------
; psdemo_large

PRO psdemo_large

; Create and initialize the component.

o = OBJ_NEW('IDLitTester')

LoadValues, o

; Create some widgets.

base = WIDGET_BASE(/COLUMN, /TLB_SIZE_EVENT, $
   TITLE = 'Property Sheet Demo (Large)')

prop = WIDGET_PROPERTYSHEET(base, value = o, $
   YSIZE = 13, /FRAME, event_pro = 'prop_event')

b1 = WIDGET_BUTTON(base, value = 'Refresh', $
   uvalue = {o:o, prop:prop}, $
   event_pro = 'refresh_event')

b2 = WIDGET_BUTTON(base, value = 'Reload', $
   uvalue = {o:o, prop:prop}, $
   event_pro = 'reload_event')

b3 = WIDGET_BUTTON(base, value = 'Hide Color', $
   uvalue = {o:o, prop:prop}, $
   event_pro = 'hide_event')

b4 = WIDGET_BUTTON(base, value = 'Show Color', $
   uvalue = {o:o, prop:prop}, $
   event_pro = 'show_event')

b5 = WIDGET_BUTTON(base, value = 'Clear', $
   uvalue = {o:o, prop:prop}, $
   event_pro = 'clear_event')

b6 = WIDGET_BUTTON(base, value = 'Desensitize', $
   uvalue = {o:o, prop:prop}, $
   event_pro = 'sensitivity_event')
```

```
b7 = WIDGET_BUTTON(base, value = 'Sensitize', $
   uvalue = {o:o, prop:prop}, $
   event_pro = 'sensitivity_event')

b8 = WIDGET_BUTTON(base, value = 'Quit', $
EVENT_PRO = 'quit_event')
; Buttons that can't be pushed after clearing:
b = [b1, b3, b4, b5, b6, b7]

; Activate the widgets.

WIDGET_CONTROL, base, SET_UVALUE = {prop:prop, b:b}, /REALIZE

XMANAGER, 'psdemo_large', base, /NO_BLOCK

END
```

The following figure displays the output of this example:

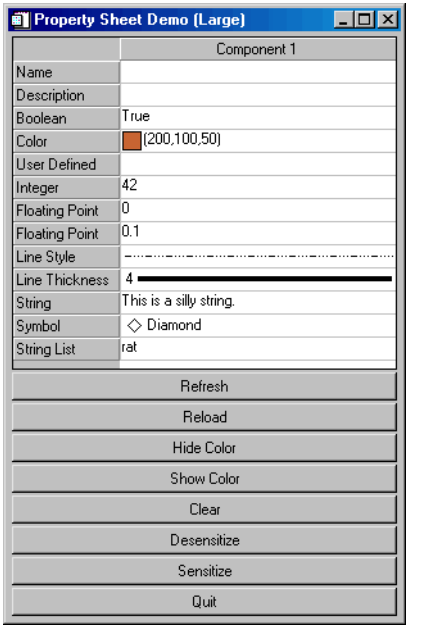


Figure 27-10: User-Defined Property Sheet Example

To demonstrate the controls available from the WIDGET_PROPERTYSHEET, do the following and note the `Selected` and `Changed` messages in the IDL Output Log:

- Change the **Boolean** field to **False**.
- Select a new **Color** from the color picker.
- Select a new “numberness” value in the **User Defined** field.
- Change the **Integer** field to a new value. Note that this field has been restricted to integers in the range -100 to 100.
- Change the first **Floating Point** field to a new value by moving the slider.
- Change the second **Floating Point** field to a new value by editing the text. Note that this field has been restricted to floating point numbers in the range -1.0 to 1.0.
- Change the **Line Style** field to a new style.
- Change the **Line Thickness** field to a new thickness.
- Select a new symbol in the **Symbol** field.
- Select a new string from the **String List**.

Click the eight buttons at the bottom of the property sheet to initiate the following events:

- The **Refresh** button loads the data specified in `refresh_event` into the property sheet, using the REFRESH_PROPERTY keyword to WIDGET_CONTROL.
- The **Reload** button reloads the data specified in `LoadValues` into the property sheet, using the SET_VALUE keyword to WIDGET_CONTROL.
- The **Hide Color** button runs `hide_event`, which sets the HIDE attribute for the color property to one.
- The **Show Color** button runs `show_event`, which sets the HIDE attribute for the color property to zero.
- The **Clear** button runs `clear_event`, which creates a new set of empty objects, deactivating all but the **Reload** button.
- The **Desensitize** button runs `sensitivity_event`, which deactivates the displayed fields.

- The **Sensitize** button runs `sensitivity_event`, which reactivates the displayed fields.
- The **Quit** button runs `quit_event`, which destroys the top-level base and ends the program.

Multiple Properties Example

The following example shows how to create a property sheet for multiple components.

Enter the following text in the IDL Editor:

```
; ExMultiSheet.pro
;
; Provides an example of a property sheet that is
; associated with more than one object. In this case,
; multiple IDLitVisAxis objects are used, with random
; colors and hidden cells, just for fun.

PRO PropertyEvent, event

IF (event.type EQ 0) THEN BEGIN   ; Value changed.

   PRINT, 'Changed: ', event.component
   PRINT, '   ', event.identifier, ': ', $
   WIDGET_INFO(event.id, COMPONENT = event.component, $
      PROPERTY_VALUE = event.identifier)

ENDIF ELSE BEGIN                ; Selection changed.

   PRINT, 'Selected: ' + event.identifier

ENDELSE

END

PRO CleanupEvent, baseID

WIDGET_CONTROL, baseID, GET_UVALUE = objects

FOR i = 0, (N_ELEMENTS(objects) - 1) DO $
   OBJ_DESTROY, objects[i]

END

PRO ExMultiSheet_event, event
```

```
ps = WIDGET_INFO(event.id, $
   FIND_BY_UNAME = 'PropSheet')

geo_tlb = WIDGET_INFO(event.id, /GEOMETRY)

WIDGET_CONTROL, ps, $
   SCR_XSIZE = geo_tlb.xsize - (2*geo_tlb.xpad), $
   SCR_YSIZE = geo_tlb.ysize - (2*geo_tlb.ypad)

END

PRO ExMultiSheet

tlb = WIDGET_BASE(/COLUMN, /TLB_SIZE_EVENTS, $
   KILL_NOTIFY = 'CleanupEvent')

; Create some columns.

oComp1 = OBJ_NEW('IDLitVisAxis', $
   COLOR = RANDOMU(s1, 3)*256, $
   TEXT_COLOR = RANDOMU(s7, 3)*256)
oComp2 = OBJ_NEW('IDLitVisAxis', $
   COLOR = RANDOMU(s2, 3)*256, $
   TEXT_COLOR = RANDOMU(s8, 3)*256)
oComp3 = OBJ_NEW('IDLitVisAxis', $
   COLOR = RANDOMU(s3, 3)*256, $
   TEXT_COLOR = RANDOMU(s9, 3)*256)
oComp4 = OBJ_NEW('IDLitVisAxis', $
   COLOR = RANDOMU(s4, 3)*256, $
   TEXT_COLOR = RANDOMU(s10, 3)*256)
oComp5 = OBJ_NEW('IDLitVisAxis', $
   COLOR = RANDOMU(s5, 3)*256, $
   TEXT_COLOR = RANDOMU(s11, 3)*256)
oComp6 = OBJ_NEW('IDLitVisAxis', $
   COLOR = RANDOMU(s6, 3)*256, $
   TEXT_COLOR = RANDOMU(s12, 3)*256)

oComps = [oComp1, oComp2, oComp3, $
   oComp4, oComp5, oComp6]

WIDGET_CONTROL, tlb, SET_UVALUE = oComps

; Hide some properties.

oComp2 -> SetPropertyAttribute, 'color', /HIDE
oComp2 -> SetPropertyAttribute, 'ticklen', /HIDE
oComp5 -> SetPropertyAttribute, 'ticklen', /HIDE

; Create the property sheet.
```

```
prop = WIDGET_PROPERTYSHEET(tlb, $
   UNAME = 'PropSheet', $
   VALUE = oComps, $
   FONT = 'Courier New*16', $
   XSIZE = 100, YSIZE = 24, $
   /FRAME, EVENT_PRO = 'PropertyEvent')

; Activate the widgets.

WIDGET_CONTROL, tlb, /REALIZE

XMANAGER, 'ExMultiSheet', tlb, /NO_BLOCK

END
```

Save the program as `ExMultiSheet.pro`, then compile and run it. A property sheet displaying the properties of six axes is displayed:

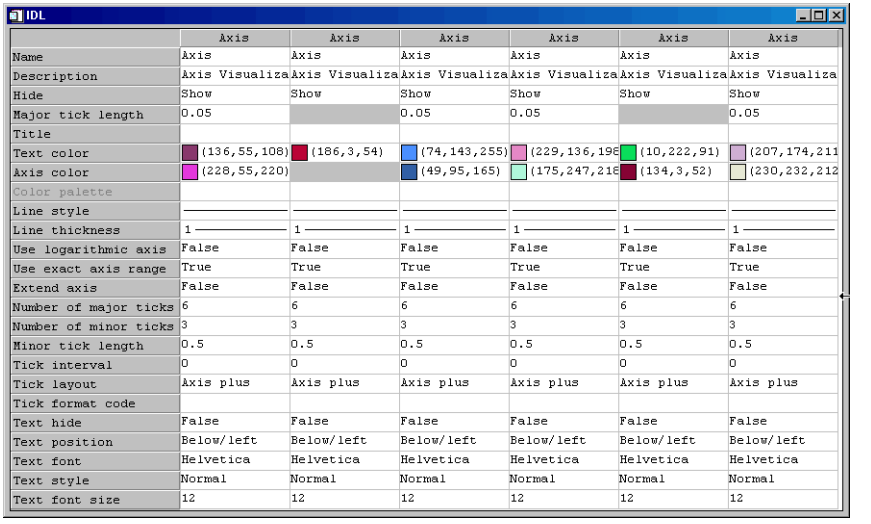


Figure 27-11: Multi-Sheet Example

The gray boxes indicate properties that have been hidden. To remove the gray boxes, comment out the code after the following comment:

```
; Hide some properties.
```

The text is displayed at 16 points in the Courier New font. To view the property sheet with the text displayed in the default size and font, comment out the following segment of the property sheet creation code:

```
FONT = "Courier New*16", $
```

To see the text displayed in a font and size of your choosing, edit the same segment to include a different font name and size.

Using Table Widgets

Table widgets display two-dimensional data and allow in-place data editing.

See "WIDGET_TABLE" in the *IDL Reference Guide* manual for a complete description of the function used to create table widgets.

This section discusses the following topics:

Default Table Size

Table widgets are sized according to the value of the following pairs of keywords to WIDGET_TABLE, in order of precedence: SCR_XSIZE/SCR_YSIZE, XSIZE/YSIZE, X_SCROLL_SIZE/Y_SCROLL_SIZE, VALUE. If either dimension remains unspecified by one of the above keywords, the default value of six (columns or rows) is used when the table is created. If the width or height specified is less than the size of the table, scroll bars are added automatically.

Note

The default row height and column width vary with different user interface toolkits.

Selection Modes

Groups of table cells can be selected either manually (using the mouse or keyboard) or programmatically. The table widget supports two selection modes — *standard* and *disjoint*. Both modes can be used either by an interactive table user or by the IDL programmer. See "Retrieving Data" on page 850 for information on retrieving data from various types of selections.

Standard Selection Mode

In standard selection mode, exactly one rectangular area (of a single cell or multiple cells) can be selected at a given time.

Interactive Selection

Interactive users select cells by clicking the left mouse button on a cell, holding the mouse button down, and dragging the mouse until the desired cells are selected. Selections can be extended by holding down the SHIFT key and selecting additional cells.

Programmatic Selection

Programmers select cells by specifying a four-element array, of the form [*left*, *top*, *right*, *bottom*], as the value of the SET_TABLE_SELECT keyword to WIDGET_CONTROL.

Disjoint Selection Mode

In disjoint selection mode, multiple rectangular areas can be selected at once. In order to place a table in disjoint selection mode, the programmer must either specify the DISJOINT_SELECTION keyword to WIDGET_TABLE when creating the table, or set the TABLE_DISJOINT_SELECTION keyword to WIDGET_CONTROL after the table has been created.

Interactive Selection

Interactive users select multiple disjoint cell regions by:

1. Creating an initial selection as described above.
2. Holding down the CONTROL key and selecting an unselected cell by clicking and holding down the left mouse button.
3. Releasing the CONTROL key (while continuing to hold the mouse button down) and dragging the mouse until the next desired region is selected.
4. Repeating as necessary.

Selections can be extended by holding down the SHIFT key and selecting additional cells.

Programmatic Selection

Programmers create select multiple disjoint cell regions by providing a 2 x *n* element array of column/row pairs specifying the cells to act upon as the value of the SET_TABLE_SELECT keyword to WIDGET_CONTROL.

Data Types

Table data can be of any IDL data type or types.

Single Data Type

If all of the table data is of the same data type, the table value is specified as a two-dimensional array.

Values returned by the GET_VALUE keyword to WIDGET_CONTROL are either a two-dimensional array (for full tables or selections when the table is in standard selection mode) or a one-dimensional array (for tables in disjoint selection mode). (See "Retrieving Data" on page 850 for details.)

Multiple Data Types

If the table contains data of several data types, the table value is specified as a vector of structures. All of the structures must be of the same type, and must contain one field for each row (if the COLUMN_MAJOR keyword to WIDGET_TABLE is set) or column (if the ROW_MAJOR keyword to WIDGET_TABLE is set; this is the default) in the table.

Values returned by the GET_VALUE keyword to WIDGET_CONTROL are either a vector of structures (for full tables or selections when the table is in standard selection mode) or a single structure with one field per cell (for selections when the table is in disjoint selection mode). (See "Retrieving Data" on page 850 for details.)

Retrieving Data

To retrieve data from a table widget, use the GET_VALUE keyword to WIDGET_CONTROL. You can retrieve the entire contents of the table or the contents of either a standard or disjoint selection. The format of the variable returned by the GET_VALUE keyword depends on the type of data displayed in the table (see "Data Types" on page 850) and the type of selection (see "Selection Modes" on page 848).

Entire Table

To retrieve data from the entire table, use the following command:

```
WIDGET_CONTROL, table, GET_VALUE=table_value
```

where *table* is the widget ID of the table widget. The *table_value* variable will contain either:

- an array with the same dimensions as the table, with one element per table cell, if the table contains data of a single data type, or
- a vector of structures, with one structure per table row or column, if the table contains structure data.

Standard Selection

To retrieve data for a group of selected cells, use the following command:

```
WIDGET_CONTROL, table, GET_VALUE=selection_value /USE_TABLE_SELECT
```

where *table* is the widget ID of the table widget. In standard selection mode, the *selection_value* variable will contain either:

- an array with the same dimensions as the selection, with one element per selected cell, if the table contains data of a single data type, or
- a vector of structures, with one structure per selected row or column, if the table contains structure data.

Note

You can also set the USE_TABLE_SELECT keyword equal to a four-element array of the form [*left*, *top*, *right*, *bottom*] containing the zero-based indices of the columns and rows that should be selected.

To retrieve the list of selected cells, use the following command:

```
selected_cells = WIDGET_INFO(table, /TABLE_SELECT)
```

where *table* is the widget ID of the table widget. The *selected_cells* variable will contain a four-element array of the form [*left*, *top*, *right*, *bottom*] containing the zero-based indices of the columns and rows that are selected.

Disjoint Selection

To retrieve data for a group of selected cells, use the following command:

```
WIDGET_CONTROL, table, GET_VALUE=selection_value, /USE_TABLE_SELECT
```

where *table* is the widget ID of the table widget. In disjoint selection mode, the *selection_value* variable will contain either:

- a one-dimensional array of values, with one element per selected cell, if the table contains data of a single data type, or
- a structure, with one field per selected cell, if the table contains structure data.

Note

You can also set the USE_TABLE_SELECT keyword equal to a 2 x *n* element array of column/row pairs specifying the cells that should be selected.

To retrieve the list of selected cells, use the following command:

```
selected_cells = WIDGET_INFO(table, /TABLE_SELECT)
```

where *table* is the widget ID of the table widget. The *selected_cells* variable will contain 2 x *n* array of column/row pairs containing the zero-based indices of the selected cells.

Converting Between Cell List Formats

With the addition of the ability to create disjoint table selections in IDL 5.6, the format of the list of selected cells returned by WIDGET_INFO was altered to accommodate non-rectangular regions when disjoint selections are enabled. To preserve backwards-compatibility, the format of the list was not changed for tables using standard selection mode, which guarantees a rectangular selection region.

If your application allows the table widget to switch between standard and disjoint selection mode, or if you have selection-handling routines that can be used with tables in either mode, you may want to modify the rectangular selection values returned for standard selections to match the lists of cells returned for disjoint selections. The following is a template for such a utility function. It accepts a four-element array of the form [*left*, *top*, *right*, *bottom*] containing the zero-based indices of the columns and rows that are selected and converts it into a 2 x *n* array of column/row pairs containing the zero-based indices the selected cells.

```
FUNCTION Make_Cell_List, Selection_Vector
  num_cells = (Selection_Vector[2]-(Selection_Vector[0]-1)) * $
    (Selection_Vector[3]-(Selection_Vector[1]-1))
  return_arr = intarr(2,num_cells)
  n=0
  FOR i=Selection_Vector[1], Selection_Vector[3] DO BEGIN
    FOR j=Selection_Vector[0], Selection_Vector[2] DO BEGIN
      return_arr(n)=j
      return_arr(n+1)=i
      n=n+2
    ENDFOR
  ENDFOR
  RETURN, return_arr
END
```

With this function compiled, you could retrieve the four-element selection array from a standard selection and turn it into a 2 x *n* element array with the following commands:

```
selected_cells = WIDGET_INFO(table, /TABLE_SELECT)
cell_list = Make_Cell_List(selected_cells)
```

where *table* is the widget ID of a table widget in standard selection mode.

To reform the array returned by

```
WIDGET_CONTROL, table, GET_VALUE=Selection_Value
```

for a standard selection into one-dimensional array like those returned for disjoint selections, use the following command:

```
REFORM(Selection_Value, N_ELEMENTS(Selection_Value), 1)
```

Edit Mode

Edit mode allows a user to select and change the contents of a table cell. There are numerous ways to enter and exit Edit mode, including:

- Clicking on an unselected cell, then typing any character. This replaces the existing text with the new character.
- Clicking on an unselected cell, then typing a carriage return. This selects the contents of the cell and positions the cursor at the right. A second carriage return exits edit mode, making no changes.
- Double-clicking on an unselected cell. This selects the contents of the cell and positions the cursor at the right.
- Clicking on a selected cell. This selects the contents of the cell and positions the cursor at the right.
- Double-clicking on a selected cell. This positions the cursor at the position where the mouse pointer was clicked.

Example: Single Data Type Data

The following procedures build a simple application that allows the user to select data from a table, plotting the data in a draw window and optionally displaying the data values in a text widget. The user can switch the table between standard and disjoint selection modes.

Note

This example is included in the file `table_widget_example1.pro` in the `examples/widgets` subdirectory of the IDL distribution. You can either open the file in an IDL editor window and compile and run the code using items on the **Run** menu, or simply enter

```
table_widget_example1
```

at the IDL command prompt. See "Running the Example Code" on page 738 if IDL does not run the program as expected.

```
; Event-handler routine
PRO table_widget_example1_event, ev

  ; Retrieve the anonymous structure contained in the user value of
  ; the top-level base widget.
  WIDGET_CONTROL, ev.top, GET_UVALUE=stash

  ; Retrieve the table's selection mode and selection.
  disjoint = WIDGET_INFO(stash.table, /TABLE_DISJOINT_SELECTION)
  selection = WIDGET_INFO(stash.table, /TABLE_SELECT)

  ; Check to see whether a selection exists, setting the
  ; variable 'hasSelection' accordingly.
  IF (selection[0] ne -1) THEN hasSelection = 1 $
    ELSE hasSelection = 0

  ; If there is a selection, get the value.
  IF (hasSelection) THEN WIDGET_CONTROL, stash.table, $
    GET_VALUE=value, /USE_TABLE_SELECT

  ; The following sections define the application's reactions to
  ; various types of events.

  ; If the event came from the table, plot the selected data.
  IF ((ev.ID eq stash.table) AND hasSelection) THEN BEGIN
    WSET, stash.draw
    PLOT, value
  ENDIF

  ; If the event came from the 'Show Selected Data' button, display
  ; the data in the text widget.
  IF ((ev.ID eq stash.b_value) AND hasSelection) THEN BEGIN
    IF (disjoint eq 0) THEN BEGIN
      WIDGET_CONTROL, stash.text, SET_VALUE=STRING(value, /PRINT)
    ENDIF ELSE BEGIN
      WIDGET_CONTROL, stash.text, SET_VALUE=STRING(value)
    ENDELSE
  ENDIF
```

```
  ; If the event came from the 'Show Selected Cells' button,
  ; display the selection information in the text widget. Use
  ; different displays for standard and disjoint selections.
  IF ((ev.ID eq stash.b_select) AND hasSelection) THEN BEGIN
    IF (disjoint eq 0) THEN BEGIN
      ; Create a string array containing the column and row
      ; values of the selected rectangle.
      list0 = 'Standard Selection'
      list1 = 'Left:   ' + STRING(selection[0])
      list2 = 'Top:    ' + STRING(selection[1])
      list3 = 'Right:  ' + STRING(selection[2])
      list4 = 'Bottom: ' + STRING(selection[3])
      list = [list0, list1, list2, list3, list4]
    ENDIF ELSE BEGIN
      ; Create a string array containing the column and row
      ; information for the selected cells.
      n = N_ELEMENTS(selection)
      list = STRARR(n/2+1)
      list[0] = 'Disjoint Selection'
      FOR j=0,n-1,2 DO BEGIN
        list[j/2+1] = 'Column: ' + STRING(selection[j]) + $
          ', Row: ' + STRING(selection[j+1])
      ENDFOR
    ENDELSE
    WIDGET_CONTROL, stash.text, SET_VALUE=list
  ENDIF

  ; If the event came from the 'Change Selection Mode' button,
  ; change the table selection mode and the title of the button.
  IF (ev.ID eq stash.b_change) THEN BEGIN
    IF (disjoint eq 0) THEN BEGIN
      WIDGET_CONTROL, stash.table, TABLE_DISJOINT_SELECTION=1
      WIDGET_CONTROL, stash.b_change, $
        SET_VALUE='Change to Standard Selection Mode'
    ENDIF ELSE BEGIN
      WIDGET_CONTROL, stash.table, TABLE_DISJOINT_SELECTION=0
      WIDGET_CONTROL, stash.b_change, $
        SET_VALUE='Change to Disjoint Selection Mode'
    ENDELSE
  ENDIF

  ; If the event came from the 'Quit' button, close the
  ; application.
  IF (ev.ID eq stash.b_quit) THEN WIDGET_CONTROL, ev.TOP, /DESTROY

END

; Widget creation routine.
```

```
PRO table_widget_example1

  ; Create data to be displayed in the table.
  data = DIST(7)

  ; Create initial text to be displayed in the text widget.
  help = ['Select data from the table below using the mouse.']

  ; Create the widget hierarchy.
  base = WIDGET_BASE(/COLUMN)
  subbase1 = WIDGET_BASE(base, /ROW)
  draw = WIDGET_DRAW(subbase1, XSIZE=250, YSIZE=250)
  subbase2 = WIDGET_BASE(subbase1, /COLUMN)
  text = WIDGET_text(subbase2, XS=50, YS=8, VALUE=help, /SCROLL)
  b_value = WIDGET_BUTTON(subbase2, VALUE='Show Selected Data')
  b_select = WIDGET_BUTTON(subbase2, VALUE='Show Selected Cells')
  b_change = WIDGET_BUTTON(subbase2, $
    VALUE='Change to Disjoint Selection Mode')
  b_quit = WIDGET_BUTTON(subbase2, VALUE='Quit')
  table = WIDGET_TABLE(base, VALUE=data, /ALL_EVENTS)

  ; Realize the widgets.
  WIDGET_CONTROL, base, /REALIZE

  ; Get the widget ID of the draw widget.
  WIDGET_CONTROL, draw, GET_VALUE=drawID

  ; Create an anonymous structure to hold widget IDs. This
  ; structure becomes the user value of the top-level base
  ; widget.
  stash = {draw:drawID, table:table, text:text, b_value:b_value, $
           b_select:b_select, b_change:b_change, b_quit:b_quit}

  ; Set the user value of the top-level base and call XMANAGER
  ; to manage everything.
  WIDGET_CONTROL, base, SET_UVALUE=stash
  XMANAGER, 'table_widget_example1', base

END
```

The following things about this example are worth noting:

- It is important to check whether a selection exists before using WIDGET_CONTROL to retrieve the selection.
- Data from disjoint selections is handled differently than data from standard selections.

- For a relatively simple application, passing a group of widget IDs via the top-level base widget's user value allows the use of a single event routine rather than separate event routines for each widget.

Example: Structure Data

The following procedures build a simple application that displays the same structure data in two table widgets; one in row-major format and one in column-major format.

Note

This example is included in the file `table_widget_example2.pro` in the `examples/widgets` subdirectory of the IDL distribution. You can either open the file in an IDL editor window and compile and run the code using items on the **Run** menu, or simply enter

```
table_widget_example2
```

at the IDL command prompt. See "Running the Example Code" on page 738 if IDL does not run the program as expected.

```
; Event-handler routine for 'Quit' button
PRO table_widget_example2_quit_event, ev
  WIDGET_CONTROL, ev.TOP, /DESTROY
END

; Widget creation routine.
PRO table_widget_example2

  ; Create some structure data.
  d0={planet:'Mercury', orbit:0.387, radius:2439, moons:0}
  d1={planet:'Venus', orbit:0.723, radius:6052, moons:0}
  d2={planet:'Earth', orbit:1.0, radius:6378, moons:1}
  d3={planet:'Mars', orbit:1.524, radius:3397, moons:2}

  ; Combine structure data into a vector of structures.
  data = [d0, d1, d2, d3]

  ; Create labels for the rows or columns of the table.
  labels = ['Planet', 'Orbit Radius (AU)', 'Radius (km)', 'Moons']

  ; To make sure the table looks nice on all platforms,
  ; set all column widths to the width of the longest string
  ; that can be a header.
  max_strlen = strlen('Orbit Radius (AU)')
  maxwidth = max_strlen * !d.x_ch_size + 6   ; ... + 6 for padding

  ; Create base widget, two tables (column- and row-major,
  ; respectively), and 'Quit' button.
```

```
  base = WIDGET_BASE(/COLUMN)
  table1 = WIDGET_TABLE(base, VALUE=data, /COLUMN_MAJOR, $
    ROW_LABELS=labels, COLUMN_LABELS='', $
    COLUMN_WIDTHS=maxwidths, /RESIZEABLE_COLUMNS)
  table2 = WIDGET_TABLE(base, VALUE=data, /ROW_MAJOR, $
    ROW_LABELS='', COLUMN_LABELS=labels, /RESIZEABLE_COLUMNS)
  b_quit = WIDGET_BUTTON(base, VALUE='Quit', $
    EVENT_PRO='table_widget_example2_quit_event')

  ; Realize the widgets.
  WIDGET_CONTROL, base, /REALIZE

  ; Retrieve the widths of the columns of the first table.
  ; Note that we must realize the widgets before retrieving
  ; this value.
  col_widths = WIDGET_INFO(table1, /COLUMN_WIDTHS)

  ; We need the following trick to get the first column (which is
  ; a header column in our first table) to reset to the width of
  ; our data columns. The initial call to keyword COLUMN_WIDTHS
  ; above only set the data column widths.
  WIDGET_CONTROL, table1, COLUMN_WIDTHS=col_widths[0], $
    USE_TABLE_SELECT=[-1,-1,3,3]
  ; This call gives table 2 the same cell dimensions as table 1
  WIDGET_CONTROL, table2, COLUMN_WIDTHS=col_widths[0], $
    USE_TABLE_SELECT=[-1,-1,3,3]

  ; Call XMANAGER to manage the widgets.
  XMANAGER, 'table_widget_example2', base

END
```

The following things about this example are worth noting:

- By default, column and row titles will contain the index of the column or row. To remove either column or row titles entirely, set the value of the COLUMN_LABELS or ROW_LABELS keyword to an empty string (`''`).
- Setting the width of the row-title column of the row-major table requires us to select column `-1` using the USE_TABLE_SELECT keyword.

Using Tab Widgets

Tab widgets create a “tabbed” interface that allows the user to select one of a list of rectangular display areas to be displayed in a single space (the *tab set*). The displayed interface elements are contained in base widgets — that is, selecting a tab displays the contents of a specified base widget within the tabbed interface. See “WIDGET_TAB” in the *IDL Reference Guide* manual for a complete description of the function used to create tab widgets.

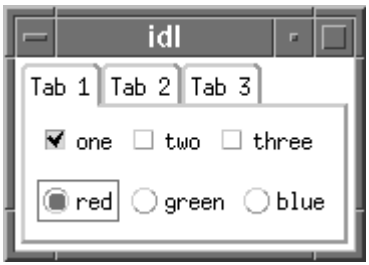


Figure 27-12: A tab widget displaying a tab set with three tabs.

This section discusses the following topics:

Example: A Simple Tab Widget

The following procedures build a simple tabbed interface with three tabs containing a variety of other widgets.

Note

This example is included in the file `tab_widget_example1.pro` in the `examples/widgets` subdirectory of the IDL distribution. You can either open the file in an IDL editor window and compile and run the code using items on the **Run** menu, or simply enter

```
tab_widget_example1
```

at the IDL command prompt. See “Running the Example Code” on page 738 if IDL does not run the program as expected.

```
; Simple event-handler routine
PRO tab_widget_example1_event, ev

  ; Retrieve the anonymous structure contained in the user value of
  ; the top-level base widget.
  WIDGET_CONTROL, ev.TOP, GET_UVALUE=stash

  ; If the user clicked the 'Done' button, destroy the widgets.
  IF (ev.ID EQ stash.bDone) THEN WIDGET_CONTROL, ev.TOP, /DESTROY

END

; Widget creation routine.
PRO tab_widget_example1, LOCATION=location

  ; Create the top-level base and the tab.
  wTLB = WIDGET_BASE(/COLUMN, /BASE_ALIGN_TOP)
  wTab = WIDGET_TAB(wTLB, LOCATION=location)

  ; Create the first tab base, containing a label and two
  ; button groups.
  wT1 = WIDGET_BASE(wTab, TITLE='TAB 1', /COLUMN)
  wLabel = WIDGET_LABEL(wT1, VALUE='Choose values')
  wBgroup1 = CW_BGROUP(wT1, ['one', 'two', 'three'], $
    /ROW, /NONEXCLUSIVE, /RETURN_NAME)
  wBgroup2 = CW_BGROUP(wT1, ['red', 'green', 'blue'], $
    /ROW, /EXCLUSIVE, /RETURN_NAME)

  ; Create the second tab base, containing a label and
  ; a slider.
  wT2 = WIDGET_BASE(wTab, TITLE='TAB 2', /COLUMN)
  wLabel = WIDGET_LABEL(wT2, VALUE='Move the Slider')
  wSlider = WIDGET_SLIDER(wT2)

  ; Create the third tab base, containing a label and
  ; a text-entry field.
  wT3 = WIDGET_BASE(wTab, TITLE='TAB 3', /COLUMN)
  wLabel = WIDGET_LABEL(wT3, VALUE='Enter some text')
  wText= WIDGET_TEXT(wT3, /EDITABLE, /ALL_EVENTS)

  ; Create a base widget to hold the 'Done' button, and
  ; the button itself.
  wControl = WIDGET_BASE(wTLB, /ROW)
  bDone = WIDGET_BUTTON(wControl, VALUE='Done')

  ; Create an anonymous structure to hold widget IDs. This
  ; structure becomes the user value of the top-level base
  ; widget.
  stash = { bDone:bDone }
```

```
    ; Realize the widgets, set the user value of the top-level
    ; base, and call XMANAGER to manage everything.
    WIDGET_CONTROL, wTLB, /REALIZE
    WIDGET_CONTROL, wTLB, SET_UVALUE=stash
    XMANAGER, 'tab_widget_example1', wTLB, /NO_BLOCK

END
```

Calling `tab_widget_example1` with the `LOCATION` keyword set to an integer value between 0 and 4 displays the same interface with the tabs placed on different sides.

As with many of the examples in this chapter, this one is designed to merely exhibit the features of the tab widget. Most of the useful things you might do with a tab widget take place in the event handling routines for the individual widgets displayed on each tab; see “Example: Retrieving Values” on page 863 for a more complicated example that stores the values of the individual widgets for later use.

Tab Sizing and Multiline Behavior

The size of the rectangular area of the tab display (where individual widgets are placed) is determined by the size of the largest base widget included in the tab set. The size of the “tab” itself (the curved area that sticks out from the rectangular base and contains the tab’s title) is determined by a number of factors, including the size of other tabs, the presence of the LOCATION and MULTILINE keywords, and the platform on which the widget application is running.

IDL attempts to create a tab that is large enough to contain the tab’s title (which is set via the TITLE keyword to WIDGET_BASE for the base widget that has the tab widget as its parent). This, coupled with the fact that the value of the MULTILINE keyword has different meanings on different platforms (see “WIDGET_TAB” in the *IDL Reference Guide* manual for details), leads to the following behaviors:

Windows Behavior

Tabs are created to show the entire text of the TITLE keyword to WIDGET_BASE.

If LOCATION = 0 or 1

Setting the LOCATION keyword to WIDGET_TAB equal to zero places the tabs on the top of the tab set; setting LOCATION to one places the tabs on the bottom of the tab set. In either case, if the MULTILINE keyword is set equal to zero, and the width of the tabs exceeds the width of the largest child base widget, the tabs are shown with scroll buttons. This allows the user to scroll through the tabs while the base widget stays immobile.

If the MULTILINE keyword is set to a positive value, the tabs will be placed in as many rows as are necessary in order to display the entire text of each tab (limited by the width of the largest base, see note below).

If LOCATION = 2 or 3

Setting the LOCATION keyword to WIDGET_TAB equal to two places the tabs on the left edge of the tab set; setting LOCATION equal to three places the tabs on the right edge of the tab set. In either case, a multiline display is always used if the width of the tabs exceeds the height of the largest child base widget, even if the MULTILINE keyword is set equal to zero. Tabs are placed in as many rows as are necessary in order to display the entire text of each tab (limited by the height of the largest base, see note below).

Note

The width or height of the tab widget is based on the width or height of the largest base widget that is a child of the tab widget. If the width of the text of one tab exceeds the width or height of the tab widget, the text will be truncated even if the MULTILINE keyword is set.

Motif Behavior

Motif platforms interpret the value of the MULTILINE keyword to be the maximum number of tabs to display per row. If the keyword is not specified or is explicitly set equal to zero, all tabs are placed on the same row. Tabs are created to show the entire text of the TITLE keyword to WIDGET_BASE. The text of the tabs is not truncated in order to make the tabs fit the space available, unless the text of a single tab exceeds the width or height of the largest base widget that is a child of the tab widget. This means that if the MULTILINE keyword is set to any value other than one, some tabs may not be displayed.

Tips for Tab Layout

There is no good way to determine in advance the best setting for the MULTILINE keyword to ensure an appropriate tab display. In most cases, however, the following suggestions should enable you to create a tab display that is useful on both Windows and UNIX platforms.

- Keep tab titles short. If you need a long description of the contents of a tab, use a label widget in the tab’s base widget rather than creating a long title.
- Set the MULTILINE keyword equal to a value greater than one. This allows you to tune the appearance of your tab set to the Motif platform without

changing the appearance under Windows, since any value greater than zero will result in a multiline tab display under Windows.

- If practical, place the tabs along the longest dimension of the tab widget, as determined by the size of the largest base widget.

Example: Retrieving Values

The following example builds on "Example: A Simple Tab Widget" on page 859 by adding the following features:

- "Next" and "Previous" buttons that switch the tab display to the next (or previous) tab in the tab set.
- A mechanism for saving the values of the widgets in the tab interface. Implementing such a mechanism allows the user to view and change all of the settings accessible via the tab widget before committing any of them.
- A mechanism for canceling — exiting from the tabbed interface without committing any changes made via the tab interface.

Note

This example is included in the file `tab_widget_example2.pro` in the `examples/widgets` subdirectory of the IDL distribution. You can either open the file in an IDL editor window and compile and run the code using items on the **Run** menu, or simply enter

```
tab_widget_example2
```

at the IDL command prompt. See "Running the Example Code" on page 738 if IDL does not run the program as expected.

```
; Main event-handler routine
PRO tab_widget_example2_event, ev

  ; Retrieve the anonymous structure contained in the user value of
  ; the top-level base widget.
  WIDGET_CONTROL, ev.TOP, GET_UVALUE=stash

  ; Retrieve the total number of tabs in the tab widget and
  ; the index of the current tab.
  numTabs = WIDGET_INFO(stash.TopTab, /TAB_NUMBER)
  thisTab = WIDGET_INFO(stash.TopTab, /TAB_CURRENT)

  ; If the current tab is the first tab, desensitize the
  ; 'Previous' button.
  IF (thisTab EQ 0) THEN BEGIN
    WIDGET_CONTROL, stash.bPrev, SENSITIVE=0
  ENDIF ELSE BEGIN
```

```
    WIDGET_CONTROL, stash.bPrev, SENSITIVE=1
  ENDELSE

  ; If the current tab is the last tab, desensitize the
  ; 'Next' button.
  IF (thisTab EQ numTabs - 1) THEN BEGIN
    WIDGET_CONTROL, stash.bNext, SENSITIVE=0
  ENDIF ELSE BEGIN
    WIDGET_CONTROL, stash.bNext, SENSITIVE=1
  ENDELSE

  ; If the user clicked either the 'Next' or 'Previous' button,
  ; cycle through the tabs by calling the 'TWE2_SwitchTab'
  ; procedure.
  IF (ev.ID EQ stash.bNext) THEN $
    TWE2_SwitchTab, thisTab, numTabs, stash, /NEXT
  IF (ev.ID EQ stash.bPrev) THEN $
    TWE2_SwitchTab, thisTab, numTabs, stash, /PREV

  ; If the user clicked the 'Done' button, print out the values of
  ; the various widgets, as contained in the 'retStruct'
  ; structure. In a real application, this step would probably
  ; adjust settings in the application to reflect the changes made
  ; by the user. Finally, destroy the widgets.
  IF (ev.ID EQ stash.bDone) THEN BEGIN
    PRINT, 'BGroup1 selected indices: ', stash.retStruct.BGROUP1
    PRINT, 'BGroup2 selected index: ', stash.retStruct.BGROUP2
    PRINT, 'Slider value: ', stash.retStruct.SLIDER
    PRINT, 'Text value: ', stash.retStruct.TEXT
    WIDGET_CONTROL, ev.TOP, /DESTROY
  ENDIF

  ; If the user clicked the 'Cancel' button, print out a message
  ; and destroy the widgets. In a real application, this step would
  ; allow the user to discard any changes made via the tabbed
  ; interface before sending them to the application.
  IF (ev.ID EQ stash.bCancel) THEN BEGIN
    PRINT, 'Update Cancelled'
    WIDGET_CONTROL, ev.TOP, /DESTROY
  ENDIF

END

; Event function to store the value of a widget in the correct
; field of the 'retStruct' structure. Note that rather than
; referring to the structure fields by name, we refer to them
; by index. This allows us to save the index value of the
; appropriate structure field in the user value of the widget
; that generates the event, which in turn allows us to use the
```

```
; same function to save the values of all of the widgets whose
; values we want to save.
;
FUNCTION TWE2_saveValue, ev
  ; Get the 'stash' structure.
  WIDGET_CONTROL, ev.TOP, GET_UVALUE=stash
  ; Get the value and user value from the widget that
  ; generated the event.
  WIDGET_CONTROL, ev.ID, GET_VALUE=val, GET_UVALUE=uval
  ; Set the value of the correct field in the 'retStruct'
  ; structure, using the field's index number (stored in
  ; the widget's user value).
  stash.retStruct.(uval) = val
  ; Reset the top-level widget's user value to the updated
  ; 'stash' structure.
  WIDGET_CONTROL, ev.TOP, SET_UVALUE=stash
END

; Procedure to cycle through the tabs when the user clicks
; the 'Next' or 'Previous' buttons.
PRO TWE2_SwitchTab, thisTab, numTabs, stash, NEXT=NEXT, PREV=PREV

  ; If user clicked the 'Next' button, we can just add one to
  ; the current tab number and use the MOD operator to cycle
  ; back to the first tab.
  IF KEYWORD_SET(NEXT) THEN nextTab = (thisTab + 1) MOD numTabs

  ; If the user clicked the 'Previous' button, we must explicitly
  ; handle the case when the user is on the first tab.
  IF KEYWORD_SET(PREV) THEN BEGIN
    IF (thisTab EQ 0) THEN BEGIN
      nextTab = numTabs - 1
    ENDIF ELSE BEGIN
      nextTab = (thisTab - 1)
    ENDELSE
  ENDIF

; Display the selected tab.
  WIDGET_CONTROL, stash.TopTab, SET_TAB_CURRENT=nextTab

END

; Widget creation routine.
PRO tab_widget_example2, LOCATION=location

  ; Create the top-level base and the tab.
  wTLB = WIDGET_BASE(/COLUMN, /BASE_ALIGN_TOP)
  wTab = WIDGET_TAB(wTLB, LOCATION=location)
```

```
; Create the first tab base, containing a label and two
; button groups. For the button groups, set the user value
; equal to the index of the field in the 'retStruct' structure
; that will hold the widget's value. Specify the
; 'TWE2_saveValue' function as the event-handler.
wT1 = WIDGET_BASE(wTab, TITLE='TAB 1', /COLUMN)
wLabel = WIDGET_LABEL(wT1, VALUE='Choose values')
wBgroup1 = CW_BGROUP(wT1, ['one', 'two', 'three'], $
  /ROW, /NONEXCLUSIVE, /RETURN_NAME, UVALUE=0, $
  EVENT_FUNC='TWE2_saveValue')
wBgroup2 = CW_BGROUP(wT1, ['red', 'green', 'blue'], $
  /ROW, /EXCLUSIVE, /RETURN_NAME, UVALUE=1, $
  EVENT_FUNC='TWE2_saveValue')

; Create the second tab base, containing a label and
; a slider. For the slider, set the user value equal
; to the index of the field in the 'retStruct' structure
; that will hold the widget's value. Specify the
; 'TWE2_saveValue' function as the event-handler.
wT2 = WIDGET_BASE(wTab, TITLE='TAB 2', /COLUMN)
wLabel = WIDGET_LABEL(wT2, VALUE='Move the Slider')
wSlider = WIDGET_SLIDER(wT2, UVALUE=2, $
  EVENT_FUNC='TWE2_saveValue')

; Create the third tab base, containing a label and
; a text-entry field. for the text widget, set the user
; value equal to the index of the field in the 'retStruct'
; structure that will hold the widget's value. Specify the
; 'TWE2_saveValue' function as the event-handler.
wT3 = WIDGET_BASE(wTab, TITLE='TAB 3', /COLUMN)
wLabel = WIDGET_LABEL(wT3, VALUE='Enter some text')
wText= WIDGET_TEXT(wT3, /EDITABLE, /ALL_EVENTS, UVALUE=3, $
  EVENT_FUNC='TWE2_saveValue')

; Create a base widget to hold the navigation and 'Done' buttons,
; and the buttons themselves. Since the first tab is displayed
; initially, make the 'Previous' button insensitive to start.
wControl = WIDGET_BASE(wTLB, /ROW)
bPrev = WIDGET_BUTTON(wControl, VALUE='<< Prev', SENSITIVE=0)
bNext = WIDGET_BUTTON(wControl, VALUE='Next >>')
bDone = WIDGET_BUTTON(wControl, VALUE='Done')
bCancel = WIDGET_BUTTON(wControl, VALUE='Cancel')

; Create an anonymous structure to hold the widget value data
; we are interested in retrieving. Note that we will refer to
; the structure fields by their indices rather than their
; names, so order is important.
retStruct={ BGROUP1:[0,0,0], BGROUP2:0, SLIDER:0L, TEXT:'empty'}
```

```
    ; Create an anonymous structure to hold widget IDs and the
    ; data structure. This structure becomes the user value of the
    ; top-level base widget.
    stash = { bDone:bDone, bCancel:bCancel, bNext:bNext, $
              bPrev:bPrev, TopTab:wTab, retStruct:retStruct}

    ; Realize the widgets, set the user value of the top-level
    ; base, and call XMANAGER to manage everything.
    WIDGET_CONTROL, wTLB, /REALIZE
    WIDGET_CONTROL, wTLB, SET_UVALUE=stash
    XMANAGER, 'tab_widget_example2', wTLB, /NO_BLOCK

END
```

The following things about this example are worth noting:

- The `retStruct` structure is an example of the kind of information you might pass *out* of a tab widget, back to a larger widget application. Using an approach like the one here allows the user to set a group of values before sending any of them to the larger application. This may be more efficient than updating the larger application “on the fly” as the user makes changes to the widgets in the tab interface.
- Similarly, if the user’s changes are not sent to the larger application until he or she clicks the “Done” button, it is important to provide a way for the user to cancel the operation entirely, without sending any changes.
- In most cases, when we refer to a field in a structure, we refer to it by its *name*. The event function `TWE2_saveValue` refers to the fields of the `retStruct` structure by their *indices* instead. We do this because while it is not possible to pass the field name in a variable, it is possible to pass the integer index value. Passing the index value of the appropriate field in the `retStruct` structure as the user value of the widget whose value is being saved allows us to write a single `TWE2_saveValue` function, rather than one function for each field in the `retStruct` structure.
- The “Next” and “Previous” buttons in this example imply that there is an order to the actions performed using the tab set. While there is no order in this example, it is easy to imagine a situation in which values from one tab would influence actions taken on another. You could even *require* that some action be taken on a given tab before a later tab could be displayed.

Using Tree Widgets

Tree widgets display information in a hierarchical structure or *tree*. Branches and sub-branches of the tree can be expanded and collapsed (either programmatically or by the user) to display or hide the information they contain. See "WIDGET_TREE" in the *IDL Reference Guide* manual for a complete description of the function used to create tree widgets.

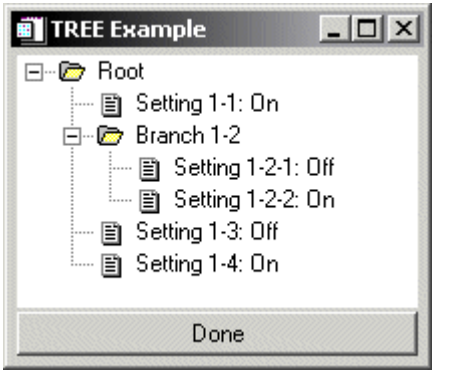


Figure 27-13: A tree widget.

This section discusses the following topics:

Types of Tree Widgets

Tree widgets behave slightly differently depending on whether their parent widget is a base widget or another tree widget:

- If the tree widget's *Parent* is a base widget, the tree widget becomes the root of a new tree widget hierarchy. The tree widget itself is not displayed, but it becomes the parent widget for other tree widgets that *are* displayed. This type

of tree widget is referred to as a *root node*. Note that a tree widget that is a root node cannot be the parent widget for any widget except another tree widget.

- If the tree widget's *Parent* is an existing tree widget, the new tree widget becomes a *branch node* or *leaf node* in the existing tree widget. In this case:
  - If the FOLDER keyword to WIDGET_TREE is present, the node becomes a *branch node*. Tree widgets that are branch nodes can become the parent widget of other tree widgets (but not of any other type of widget).
  - If the FOLDER keyword to WIDGET_TREE is *not* present, the node becomes a *leaf node*. Leaf nodes cannot serve as the parent widget for any other widget.

Example: A Simple Tree

The following example builds a simple tree widget (shown in Figure 27-13). Double-clicking on the leaf nodes toggles the value of the displayed text between the values "On" and "Off."

Note

This example is included in the file `tree_widget_example.pro` in the `examples/widgets` subdirectory of the IDL distribution. You can either open the file in an IDL editor window and compile and run the code using items on the **Run** menu, or simply enter

```
tree_widget_example
```

at the IDL command prompt. See "Running the Example Code" on page 738 if IDL does not run the program as expected.

```
; Event handler routine.
PRO tree_widget_example_event, ev

  ; We use widget user values to determine what action to take.
  ; First retrieve the user value (if any) from the widget that
  ; generated the event.

  WIDGET_CONTROL, ev.ID, GET_UVALUE=uName

  ; If the widget that generated the event has a user value, check
  ; its value and take the appropriate action.
  IF (N_ELEMENTS(uName) NE 0) THEN BEGIN
    IF (uName EQ 'LEAF') THEN BEGIN
      ; Make sure the value does not change when the leaf node
      ; is selected with a single click.
      IF (ev.CLICKS EQ 2) THEN TWE_ToggleValue, ev.ID
    ENDIF
```

```
    IF (uName EQ 'DONE') THEN WIDGET_CONTROL, ev.TOP, /DESTROY
  ENDIF

END

; Routine to change the value of a leaf node's text.
PRO TWE_ToggleValue, widID

  ; Get the current value.
  WIDGET_CONTROL, widID, GET_VALUE=curVal

  ; Split the string at the colon character.
  full_string = STRSPLIT(curVal, ':', /EXTRACT)

  ; Check the value of the text after the colon, and toggle
  ; to the new value.
  full_string[1] = (full_string[1] EQ ' Off') ? ': On' : ': Off'

  ; Reset the value of the leaf node's text.
  WIDGET_CONTROL, widID, SET_VALUE=STRJOIN(full_string)

END

; Widget creation routine.
PRO tree_widget_example

  ; Start with a base widget.
  wTLB = WIDGET_BASE(/COLUMN, TITLE='Tree Example')

  ; The first tree widget has the top-level base as its parent.
  ; The visible tree widget branches and leaves will all be
  ; descendants of this tree widget.
  wTree = WIDGET_TREE(wTLB)

  ; Place a folder at the root level of the visible tree.
  wtRoot = WIDGET_TREE(wTree, VALUE='Root', /FOLDER, /EXPANDED)

  ; Create leaves and branches. Note that we set the user value of
  ; every leaf node (tree widgets that do not have the FOLDER
  ; keyword set) equal to 'LEAF'. We use this in the event handler
  ; to determine whether or not to change the text value.
  wtLeaf11 = WIDGET_TREE(wtRoot, VALUE='Setting 1-1: Off', $
    UVALUE='LEAF')
  wtBranch12 = WIDGET_TREE(wtRoot, VALUE='Branch 1-2', $
    /FOLDER, /EXPANDED)
  wtLeaf121 =WIDGET_TREE(wtBranch12, VALUE='Setting 1-2-1: Off', $
    UVALUE='LEAF')
  wtLeaf122 =WIDGET_TREE(wtBranch12, VALUE='Setting 1-2-2: Off', $
    UVALUE='LEAF')
```

```
  wtLeaf13 = WIDGET_TREE(wtRoot, VALUE='Setting 1-3: Off', $
    UVALUE='LEAF')
  wtLeaf14 = WIDGET_TREE(wtRoot, VALUE='Setting 1-4: Off', $
    UVALUE='LEAF')

  ; Create a 'Done' button, setting its user value for use in the
  ; event handler.
  wDone = WIDGET_BUTTON(wTLB, VALUE="Done", UVALUE='DONE')

  ; Realize the widgets and run XMANAGER to manage them.
  WIDGET_CONTROL, wTLB, /REALIZE
  XMANAGER, 'tree_widget_example', wTLB, /NO_BLOCK

END
```

As with many of the examples in this chapter, this one is designed to merely exhibit the features of the tree widget. Most of the useful things you might do with a tree widget take place in the event handling routines for the leaf nodes; whereas in this example clicking on a leaf simply changes the displayed text value, in a real application more complicated things might take place. Alternately, you might use a tree widget for display purposes only, in which case user interaction would be limited to expanding and collapsing the branches.

Setting the Tree Selection State

You can programmatically select or deselect nodes in a tree widget hierarchy using the SET_TREE_SELECT keyword to WIDGET_CONTROL. Selecting a node or nodes visually highlights the node on the tree display. In the above example, placing the following command just above the call to XMANAGER:

```
WIDGET_CONTROL, wtLeaf11, /SET_TREE_SELECT
```

would cause the first leaf node to be highlighted when the widget tree was first displayed.

Making a Tree Entry Visible

If you tree is large or has many branches, you may need to explicitly bring a given node to the user's attention. You can do this using the SET_TREE_VISIBLE keyword to WIDGET_CONTROL:

```
WIDGET_CONTROL, wTreeNode, /SET_TREE_VISIBLE
```

were *wTreeNode* is any node attached to a tree widget — that is, any tree widget that has another tree widget as its parent widget. Setting this keyword has two possible effects:

1. If the specified node is inside a collapsed folder, the folder and all folders above it are expanded to reveal the node.
2. If the specified node is in a portion of the tree that is not currently visible because the tree has scrolled within the parent base widget, the tree view scrolls so that the selected node is at the top of the base widget.

Use of this keyword does not affect the tree widget selection state.

Replacing the Default Bitmaps

By default, tree widgets use bitmap images of a folder and a single piece of paper as the icons representing branch and leaf nodes in a tree widget hierarchy. You can modify the look of the tree widget by supplying your own bitmap for a given node. Set the BITMAP keyword to WIDGET_TREE equal to a 16 x 16 x 3 array that contains a 24-bit color image.

For example, suppose you have a 16 x 16 pixel TrueColor icon stored in a TIFF file. The following commands make the image the icon used for the root node of a tree widget hierarchy:

```
myIcon = READ_TIFF('/path_to/myicon.tif', INTERLEAVE=2)
wtRoot = WIDGET_TREE(wTree, /FOLDER, BITMAP=myIcon)
```

Note the use of the INTERLEAVE keyword to ensure that the resulting image array has dimensions 16 x 16 x 3. Depending on your image file format, you may need to modify the image array in other ways.

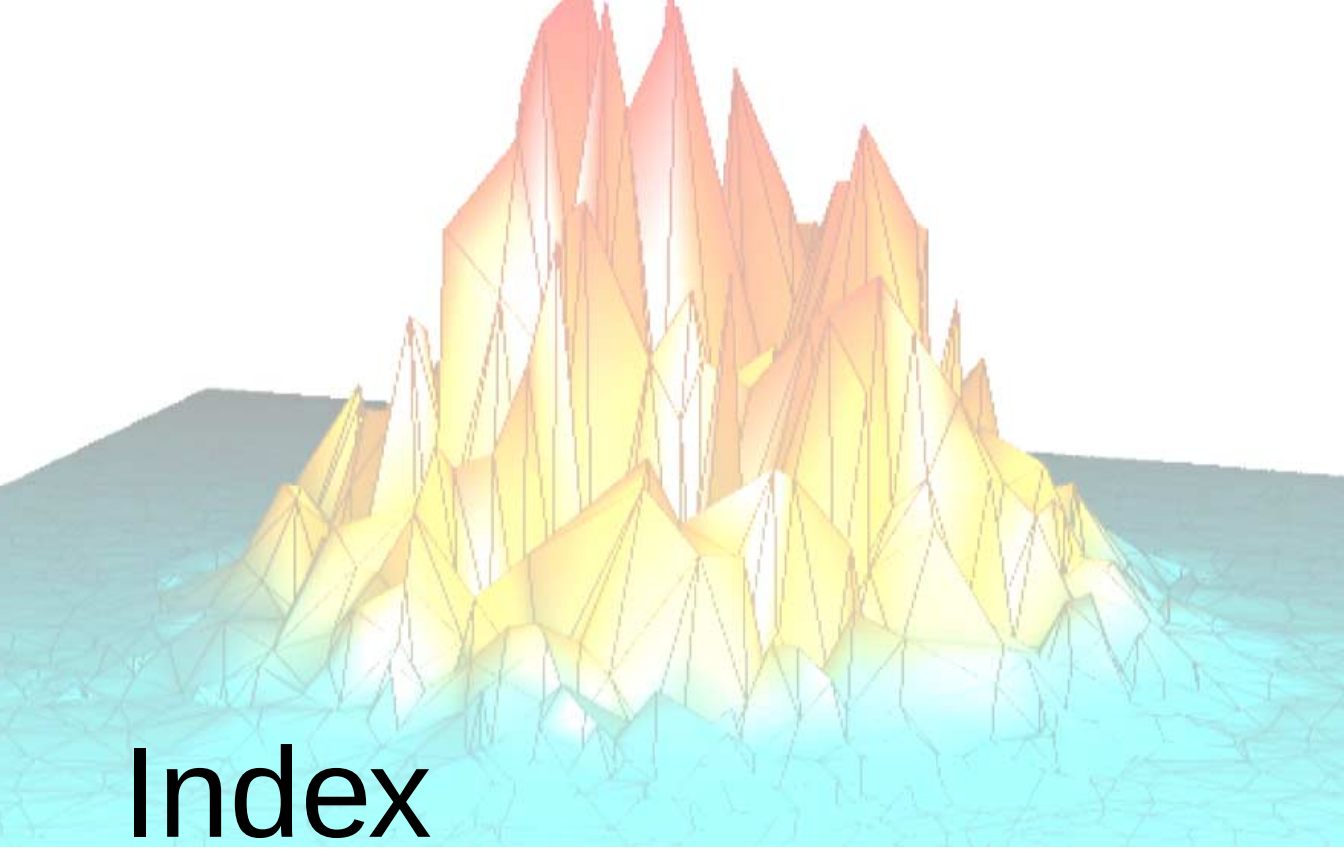
Using Images from the IDL Distribution

The `/resources/bitmaps` subdirectory of the IDL distribution contains a selection of 16-color (4-bit), 16 x 16 pixel icon images. To use these images as bitmaps in a tree widget, you must convert them to 16 x 16 x 3 (24-bit color) arrays.

The following code snippet loads the camera icon stored in `IDLDIR/resources/bitmaps/camera.bmp` into a 16 x 16 x 3 array:

```
; Create a 24-bit image array.
imageRGB = BYTARR(16,16,3,/NOZERO)
; Read in the bitmap.
file=FILEPATH('camera.bmp', SUBDIR=['resource', 'bitmaps'])
image8 = READ_BMP(file, Red, Green, Blue)
; Pass the image through the color table
imageRGB[0,0,0] = Red[image8]
imageRGB[0,0,1] = Green[image8]
imageRGB[0,0,2] = Blue[image8]
```

To use the camera icon in a tree widget, you would specify the `imageRGB` variable as the value of the BITMAP keyword to WIDGET_TREE.



Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z