



Chapter 12: Procedures and Functions

The following topics are covered in this chapter:

Overview	290	Entering Procedure Definitions	306
Defining a Procedure	291	How IDL Resolves Routines	308
Defining a Function	293	Parameter Passing Mechanism	309
Parameters	296	Calling Mechanism	311
Using Keyword Parameters	299	Setting Compilation Options	313
Keyword Inheritance	301	Advice for Library Authors	315

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 293.
- 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 299.

Automatic Execution

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

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

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

Note

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

Warning

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

For information on how to access routines, see [“Executing Program Files”](#) in Chapter 2 of the *Using IDL* manual.

Forward Function Definition

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

This problem has been addressed beginning with IDL version 5.0 by the use of square brackets “[]” instead of parentheses to specify array subscripts. See [“Array Subscript Syntax: \[\] vs. \(\)”](#) on page 86 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 be abbreviated to the shortest unambiguous string. The last example sets one of the three keywords according to the value of the variable I.

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

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

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

Keyword Inheritance

Keyword inheritance allows IDL routines to accept keyword parameters not defined in their function or procedure declaration and pass them on to routines they call. This greatly simplifies writing “wrapper” routines, which are variations of a system or user-provided routine. Specifically, keyword inheritance allows your routines to accept keywords accepted by routines that it calls without explicitly handling each keyword individually.

There are two distinct mechanisms to handle keyword inheritance: one to pass keyword parameters by *value*, and another to pass keyword parameters by *reference*.

`_EXTRA`: Passing Keyword Parameters by Value

You can pass keyword parameters to called routines by *value* by adding the formal keyword parameter “`_EXTRA`” (note the underscore character) to the definition of your routine. Passing parameters by value means that you are giving the called routine the *contents* of an existing IDL variable to work with. In turn, this means that keyword parameters passed into a routine by value cannot be returned to the calling routine — there is no variable name into which the value can be placed.

When a routine is defined with the formal keyword parameter `_EXTRA`, pairs of unrecognized keywords and values are placed in an anonymous structure. The name of each unrecognized keyword becomes a tag name, and the keyword value becomes the tag value. Changes to this structure created by using the `_EXTRA` keyword do not affect variables in the calling program.

When the keyword `_EXTRA` appears in a procedure or function call, its argument is either a structure containing additional keyword/value pairs which are inserted into the argument list, or a string array as described in the next section. The value of `_EXTRA` can also be “undefined”, indicating that no additional keyword parameters were passed.

`_REF_EXTRA`: Passing Keyword Parameters by Reference

You can pass keyword parameters to called routines by *reference* by adding the formal keyword parameter “`_REF_EXTRA`” (note the underscore character) to the definition of your routine. Passing parameters by reference means that you are giving the called routine the *name* of an existing IDL variable to work with; IDL takes care of keeping track of the value associated with the name. The *values* of keyword parameters specified via `_REF_EXTRA` are *not* available to the routine that is passing the keywords on.

When a routine is defined with the formal keyword parameter `_REF_EXTRA`, pairs of unrecognized keywords and values are placed in a storage location that is accessible to both calling and called routines, and the keyword names are placed in an IDL string array. The string array can be “deciphered” using the `_EXTRA` keyword, which matches the names in the string with the “live” values in the storage location. This means that if the keywords specify IDL variables, the values of those variables can be altered by any routine that has access to the variable via the keyword inheritance mechanism. In this fashion, the values of keyword parameters can be changed within a routine and passed back to the routine’s caller.

The “pass by reference” keyword inheritance mechanism is especially useful when writing object methods, which may be inherited multiple times and which often wish to change the value of variables available to the calling method. (The values of object properties are one example of data that can profitably be shared by objects at various levels in an object hierarchy.)

Accepting Extra Keyword Parameters

While you must choose whether a routine will *pass* extra keyword parameters by value or by reference when defining the routine (specifying both `_EXTRA` and `_REF_EXTRA` as formal parameters will cause an error), routines that *accept* extra keyword parameters can use either the `_EXTRA` keyword or the `_REF_EXTRA` keyword. However, it is not possible to both have access to the keyword values and pass them along to called routines by reference within the same routine. This means that any routine that needs access to the passed keyword parameters must use `_EXTRA` in its definition statement, or define the keyword explicitly itself.

Selective Keyword Redirection

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

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

If we call the `SOMEPROC` routine with three keywords:

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

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

- it will pass the keyword SQUIRREL at its value to procedure TWO,
- it will do nothing with the keyword SPY.

Choosing a Keyword Inheritance Mechanism

The “pass by reference” (`_REF_EXTRA`) keyword inheritance mechanism was introduced in IDL version 5.1, and in many cases is a good choice even if values are not being passed back to the calling routine. Because the `_REF_EXTRA` mechanism does not create an IDL structure to hold the keyword/value pairs, overhead is slightly reduced. Two situations lend themselves to use of the `_REF_EXTRA` mechanism:

1. You need to pass the values of keyword variables back from a called routine to the calling routine.
2. Your routine is an “inner loop” routine that may be called many times. If the routine is called repeatedly, the savings resulting from not creating a new IDL structure with each call may be significant.

It is important to remember that if the routine that is passing the keyword values through also needs access to the values of the keywords for some reason, you must use the “pass by value” (`_EXTRA`) mechanism.

Note

Updating existing routines that use `_EXTRA` to use `_REF_EXTRA` is relatively easy. Since the called routine uses `_EXTRA` to receive the extra keywords in either case, you need only change the `_EXTRA` to `_REF_EXTRA` in the definition of the calling routine.

By contrast, the “pass by value” (`_EXTRA`) keyword inheritance mechanism is useful in the following situations:

1. Your routine needs access to the values of the extra keywords for some reason.
2. You want to ensure that variables specified as keyword parameters are not changed by a called routine.

Example: Keywords Passed by Value

One of the most common uses for the “pass by value” keyword inheritance mechanism is to create “wrapper” routines that extend the functionality of existing routines. In most “wrapper” routines, there is no need to return values to the calling routine — the aim is simply to implement the complete set of keywords available to the existing routine in the wrapper routine.

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

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

The procedure definition:

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

places unrecognized keywords (e.g., any keywords other than COLOR) and their values into the variable “e”. If there are no unrecognized keywords, e will be undefined.

When procedure TEST is called with the following command:

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

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

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

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

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

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

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

specifies a color index of 12 to PLOT.

Example: Keywords Passed by Reference

The “pass by reference” keyword inheritance mechanism allows you to change the value of a variable in the calling routine’s context from within the routine. To demonstrate the difference between _EXTRA and _REF_EXTRA, consider the following simple example procedures:

```
PRO TEST1, _EXTRA = ex
HELP, _EXTRA = ex
END

PRO TEST2, _REF_EXTRA = ex
HELP, _EXTRA = ex
END
```

Both TEST1 and TEST2 are simple wrappers to the HELP procedure. Observe the result when we call each routine, specifying OUTPUT as an extra keyword parameter, then use the HELP procedure again to determine the value of the output variable:

```
TEST1, OUTPUT = out & HELP, out
```

IDL prints:

```
% At TEST1                2 /dev/tty
% $MAIN$
EX UNDEFINED = <Undefined>
Compiled Procedures:
    $MAIN$ TEST1
Compiled Functions:
```

Now run TEST2:

```
TEST2, OUTPUT = out & HELP, out
```

IDL prints:

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

Entering Procedure Definitions

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

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

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

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

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

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

Note Regarding Functions

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

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

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

Note

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

work as in previous version of IDL, we strongly suggest that you use brackets in all new code. See [“Array Subscript Syntax: \[\] vs. \(\)”](#) on page 86 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 294.
- 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 parenthesis to indicate either functions or arrays) or that the desired routine does not exist (See [Chapter 5, “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 105 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.
```