

## User-Defined Functions

Overview .....	28
Function Syntax.....	29
The Function Name .....	29
The Procedure Subtype.....	29
The Parameter List and Parameter Declarations.....	30
Optional Parameters.....	30
Local Variable Declarations.....	30
Body Code.....	31
The Return Statement.....	31
Conditional Statements in Functions .....	31
If-Else-Endif .....	31
If-Elseif-Endif.....	32
Comparisons.....	32
Bitwise and Logical Operators.....	33
Using Bitwise Operators.....	33
Switch Statements .....	34
Loops.....	36
Do-While Loop.....	36
Nested Do-While Loops.....	36
While Loop .....	36
For Loop .....	37
Break Statement .....	37
Continue Statement .....	38
Flow Control for Aborts.....	38
AbortOnRTE Keyword .....	38
AbortOnValue Keyword .....	38
try-catch-endtry Flow Control.....	38
try-catch-endtry Example.....	39
Constants.....	40
Pragmas.....	40
The rtGlobals Pragma .....	41
The version Pragma.....	42
The IgorVersion Pragma.....	42
The ModuleName Pragma .....	42
The IndependentModule Pragma .....	43
Unknown Pragmas.....	43
Proc Pictures .....	43
How Parameters Work.....	44
Example of Pass-By-Value.....	44
Pass-By-Reference.....	45
How Waves Are Passed.....	46
Using Optional Parameters .....	46
Local Versus Global Variables .....	46
Local Variables Used by Igor Operations.....	47

## Chapter IV-3 — User-Defined Functions

---

Converting a String into a Reference Using \$ .....	47
Using \$ to Refer to a Window .....	48
Using \$ In a Data Folder Path .....	49
Compile Time Versus Runtime .....	49
Accessing Global Variables and Waves .....	50
Runtime Lookup of Globals .....	50
Put WAVE Declaration After Wave Is Created .....	52
Runtime Lookup Failure .....	52
Runtime Lookup Failure and the Debugger .....	52
Accessing Complex Global Variables and Waves .....	53
Accessing Text Waves .....	53
Accessing Global Variables and Waves Using Liberal Names .....	53
Runtime Lookup Example .....	54
Automatic Creation of NVAR and SVAR References .....	55
Wave References .....	56
Automatic Creation of WAVE References .....	56
Standalone WAVE Reference Statements .....	57
Inline WAVE Reference Statements .....	57
WAVE Reference Types .....	58
WAVE Reference Type Flags .....	58
Problems with Automatic Creation of WAVE References .....	59
WAVE Reference Is Not Needed to Pass a Wave Parameter .....	60
Wave Reference Function Results .....	60
Wave Reference Waves .....	60
Data Folder References .....	61
Using Data Folder References .....	62
The /SDFR Flag .....	63
The DFREF Type .....	63
Built-in DFREF Functions .....	64
Checking Data Folder Reference Validity .....	64
Data Folder Reference Function Results .....	64
Data Folder Reference Waves .....	65
Accessing Waves in Functions .....	65
Wave Reference Passed as Parameter .....	66
Wave Accessed Via String Passed as Parameter .....	66
Wave Accessed Via String Calculated in Function .....	67
Wave Accessed Via Literal Wave Name .....	67
Wave Accessed Via Wave Reference Function .....	68
Destination Wave Parameters .....	68
Wave Reference as Destination Wave .....	69
Exceptions To Destination Wave Rules .....	69
Updating of Destination Wave References .....	69
Inline Wave References With Destination Waves .....	69
Destination Wave Reference Issues .....	70
Changes in Destination Wave Behavior .....	70
Trace Name Parameters .....	71
User-defined Trace Names .....	71
Free Waves .....	71
Free Wave Created When Free Data Folder Is Deleted .....	72
Free Wave Created For User Function Input Parameter .....	73
Free Wave Lifetime .....	73
Converting a Free Wave to a Global Wave .....	75
Free Data Folders .....	75
Free Data Folder Lifetime .....	76
Free Data Folder Objects Lifetime .....	77
Converting a Free Data Folder to a Global Data Folder .....	78

Structures in Functions .....	78
Defining Structures.....	78
Using Structures.....	79
Example.....	80
Built-In Structures.....	81
Applications of Structures .....	82
Using Structures with Windows and Controls .....	82
Example.....	82
Limitations of Structures .....	83
Static Functions .....	83
ThreadSafe Functions .....	83
Function Overrides .....	84
Function References.....	84
Conditional Compilation .....	86
Predefined Global Symbols.....	87
Conditional Compilation Examples.....	87
Function Errors.....	88
Coercion in Functions.....	88
Operations in Functions.....	89
Updates During Function Execution .....	89
Aborting Functions .....	89
Legacy Code Issues.....	90
Old-Style Comments and Compatibility Mode .....	90
Text After Flow Control.....	91
Global Variables Used by Igor Operations .....	91
Direct Reference to Globals .....	91

### Overview

Most of Igor programming consists of writing user-defined functions.

A function has zero or more numeric, string and wave parameters. You can use local variables to store intermediate results. The function body consists of Igor operations, assignment statements, flow control statements, and calls to other functions.

A function can return a numeric or string result. It can also have a side-effect, such as creating a wave or creating a graph.

Before we dive into the technical details, here is an informal look at some simple examples.

```
Function Hypotenuse(side1, side2)
    Variable side1, side2

    Variable hyp
    hyp = sqrt(side1^2 + side2^2)

    return hyp
End
```

The Hypotenuse function takes two numeric parameters and returns a numeric result. “hyp” is a local variable and sqrt is a built-in function. You could test Hypotenuse by executing the following statement in the command line:

```
Print Hypotenuse(3, 4)
```

Now let’s look at a function that deals with text strings.

```
Function/S FirstStr(str1, str2)
    String str1, str2

    String result

    if (CmpStr(str1, str2) < 0)
        result = str1
    else
        result = str2
    endif

    return result
End
```

The FirstStr function takes two string parameters and returns the string that is first in alphabetical order. CmpStr is a built-in function. You could test FirstStr by executing the following statement in the command line:

```
Print FirstStr("ABC", "BCD")
```

Now a function that deals with waves.

```
Function CreateRatioOfWaves(w1, w2, nameOfOutputWave)
    WAVE w1, w2
    String nameOfOutputWave

    Duplicate/O w1, $nameOfOutputWave
    WAVE wOut = $nameOfOutputWave
    wOut = w1 / w2
End
```

The CreateRatioOfWaves function takes two wave parameters and a string parameter. The string is the name to use for a new wave, created by duplicating one of the input waves. The “WAVE wOut” statement creates a wave reference for use in the following assignment statement. This function has no direct result (no return statement) but has the side-effect of creating a new wave.

Here are some commands to test CreateRatioOfWaves:

```
Make test1 = {1, 2, 3}, test2 = {2, 3, 4}
CreateRatioOfWaves(test1, test2, "ratio")
Edit test1, test2, ratio
```

## Function Syntax

The basic syntax of a function is:

```
Function <Name> (<Parameter list> [<Optional Parameters>]) [:<Subtype>]
  <Parameter declarations>

  <Local variable declarations>

  <Body code>

  <Return statement>
End
```

Here is an example:

```
Function Hypotenuse(side1, side2)
  Variable side1, side2          // Parameter declaration

  Variable hyp                   // Local variable declaration

  hyp = sqrt( side1^2 + side2^2 ) // Body code

  return hyp                     // Return statement
End
```

You could test this function from the command line using one of these commands:

```
Print Hypotenuse(3,4)
Variable/G result = Hypotenuse(3,4); Print result
```

As shown above, the function returns a real, numeric result. The Function keyword can be followed by a flag that specifies a different result type.

Flag	Return Value Type
/C	Complex number
/S	String
/D	Double precision number (obsolete)

The /D flag is obsolete because all calculations are now performed in double precision. However, it is still permitted.

### The Function Name

The names of functions must follow the standard Igor naming conventions. Names can consist of up to 31 characters. The first character must be alphabetic while the remaining characters can include alphabetic and numeric characters and the underscore character. Names must not conflict with the names of other Igor objects, functions or operations. Names in Igor are case insensitive.

### The Procedure Subtype

You can identify procedures designed for specific purposes by using a subtype. Here is an example:

```
Function ButtonProc(ctrlName) : ButtonControl
  String ctrlName
```

## Chapter IV-3 — User-Defined Functions

---

```
    Beep
End
```

Here, “ : ButtonControl” identifies a function intended to be called when a user-defined button control is clicked. Because of the subtype, this function is added to the menu of procedures that appears in the Button Control dialog. When Igor automatically generates a procedure it generates the appropriate subtype. See **Procedure Subtypes** on page IV-179 for details.

### The Parameter List and Parameter Declarations

The parameter list specifies the name for each input parameter. There is no limit on the number of parameters.

All parameters must be declared immediately after the function declaration.

The parameter declaration must declare the type of each parameter using the keywords *Variable*, *String*, *WAVE* or *DFREF*.

If a parameter is a complex number, it must be declared *Variable/C*.

If it is a complex wave, it must be declared *WAVE/C*.

If it is a text wave, it must be declared *WAVE/T*.

Variable and string parameters are usually passed to a subroutine *by value* but can also be passed *by reference*. For an explanation of these terms, see **How Parameters Work** on page IV-44.

### Optional Parameters

Following the list of required function input parameters, you can also specify a list of optional input parameters by enclosing the parameter names in brackets. You can supply any number of optional parameter values when calling the function by using the *ParamName=Value* syntax. Optional parameters may be of any valid data type. There is no limit on the number of parameters.

All optional parameters must be declared immediately after the function declaration. As with all other variables, optional parameters are initialized to zero. You must use the **ParamIsDefault** function to determine if a particular optional parameter was supplied in the function call.

See **Using Optional Parameters** on page IV-46 for an example.

### Local Variable Declarations

The parameter declarations are followed by the local variable declarations if the procedure uses local variables. Local variables exist only during the execution of the procedure. They can be numeric or string and are declared using the *Variable* or *String* keywords. They can optionally be initialized. Here is an example:

```
Function Example(p1)
    Variable p1

    // Here are the local variables
    Variable v1, v2
    Variable v3=0
    Variable/C cv1=cplx(0,0)
    String s1="test", s2="test2"

    <Body code>
End
```

If you do not supply explicit initialization, Igor automatically initializes local numeric variables with the value zero. Local string variables are initialized with a null value such that, if you try to use the string before you store a value in it, Igor will report an error.

The name of a local variable is allowed to conflict with other names in Igor although they must be unique within the function. Clearly if you create a local variable named “sin” then you will be unable to use Igor’s built-in sin function within the function.

You can declare local variables in the body code section of a function as well as in the local variable section.

As of Igor Pro 5, you can define structures and use them as local variables in functions. Structures are defined outside of function definitions and are instantiated as local variables using the STRUCT keyword as described in **Structures in Functions** on page IV-78.

### Body Code

This table shows what can appear in body code of a function.

What	Allowed in Functions?	Comment
Assignment statements	Yes	Includes wave, variable and string assignments.
Built-in operations	Yes, with a few exceptions.	See <b>Operations in Functions</b> on page IV-89 for exceptions.
Calls to user functions	Yes	
Calls to macros	No	
External functions	Yes	
External operations	Yes, with exceptions.	

As of Igor Pro 5, external operations can be designed to be directly callable from user-defined functions. External operations implemented by old XOPs can not be called directly.

The Execute operation provides a way to work around the limitation on what can be called from user functions. See **The Execute Operation** on page IV-176.

### The Return Statement

A return statement often appears at the end of a function, but it can appear anywhere in the function body. You can also have more than one return statement.

The return statement immediately stops executing the function and returns a value to the calling function. If the function is declared as a string function (Function/S), then the return statement must return a string. Otherwise it must return a number.

If there is no return statement or if a function ends without hitting a return statement then the function returns the value NaN (Not a Number) for numeric functions and null for string functions. If the calling function attempts to use the null value, Igor will report an error.

## Conditional Statements in Functions

Igor Pro supports two basic forms of conditional statements: if-else-endif and if-elseif-endif statements. Igor also supports multiway branching with switch and strswitch statements.

### If-Else-Endif

The form of the if-else-endif structure is

```
if ( <expression> )
  <TRUE part>
else
  <FALSE part>
endif
```

## Chapter IV-3 — User-Defined Functions

---

<expression> is a numeric expression that is considered TRUE if it evaluates to any nonzero number and FALSE if it evaluates to zero. The TRUE part and the FALSE part may consist of any number of lines. If the expression evaluates to TRUE, only the TRUE part is executed and the FALSE part is skipped. If the expression evaluates to FALSE, only the FALSE part is executed and the true part is skipped. After the TRUE part or FALSE part code is executed, execution will continue with any code immediately following the if-else-endif statement.

The keyword “else” and the FALSE part may be omitted to give a simple conditional:

```
if ( <expression> )
    <TRUE part>
endif
```

Because Igor is line-oriented, you may not put the `if`, `else` and `endif` keywords all on the same line. They must each be in separate lines with no other code.

### If-Elseif-Endif

The if-elseif-endif statement provides a means for creating nested if structures. It has the form:

```
if ( <expression1> )
    <TRUE part 1>
elseif ( <expression2> )
    <TRUE part 2>
[else
    <FALSE part>]
endif
```

These statements follow similar rules as for if-else-endif statements. When any expression evaluates as TRUE (nonzero) the code immediately following the expression will be executed. If all expressions evaluate as FALSE (zero) and there is an else clause, then the statements following the else keyword will be executed. Once any code in a TRUE part or the FALSE part is executed, execution will continue with any code immediately following the if-elseif-endif statement.

### Comparisons

The relational comparison operators are used in numeric conditional expressions.

Symbol	Meaning	Symbol	Meaning
==	equal	<=	less than or equal
!=	not-equal	>	greater than
<	less than	>=	greater than or equal

These operators return 1 for TRUE and 0 for FALSE.

The comparison operators work with numeric operands only. To do comparisons with string operands, use the **cmpstr** function.

Comparison operators are usually used in conditional structures but can also be used in arithmetic expressions. For example:

```
wave0 = wave0 * (wave0 < 0)
```

This clips all positive values in wave0 to zero.

See also **Example: Comparison Operators and Wave Synthesis** on page II-98.

## Bitwise and Logical Operators

The bitwise and logical operators are also used in conditional expressions.

Symbol	Meaning	Symbol	Meaning
~	Bitwise complement	!	Logical NOT
&	Bitwise AND	&&	Logical AND
	Bitwise OR		Logical OR

The precedence of operators is shown in the table under **Operators** on page IV-5. In the absence of parentheses, an operator with higher precedence (higher in the table) is executed before an operator with lower precedence.

Because the precedence of the arithmetic operators is higher than the precedence of the comparison operators, you can write the following without parentheses:

```
if (a+b != c+d)
    Print "a+b is not equal to c+d"
endif
```

Because the precedence of the comparison operators is higher than the precedence of the logical OR operator, you can write the following without parentheses:

```
if (a==b || c==d)
    Print "Either a equals b or c equals d"
endif
```

For operators with the same precedence, there is no guaranteed order of execution and you must use parentheses to be sure of what will be executed. For example:

```
if ((a&b) != (c&d))
    Print "a ANDED with b is not equal to c ANDED with d"
endif
```

See **Operators** on page IV-5 for more discussion of operators.

### Using Bitwise Operators

The bitwise operators are used to test, set, and clear bits. This makes sense only when you are dealing with integer operands.

Bit Action	Operation
Test	AND operator (&)
Set	OR operator ( )
Clear	Bitwise complement operator (~) followed by the bitwise AND operator (&)
Shift left	Multiply by powers of 2
Shift right	Divide by powers of 2

This function illustrates various bit manipulation techniques.

```
Function DemoBitManipulation(vIn)
    Variable vIn

    vIn = trunc(vIn) // Makes sense with integers only
    Printf "Original value: %d\r", vIn

    Variable vOut
```

## Chapter IV-3 — User-Defined Functions

---

```
if ((vIn & 2^3) != 0)           // Test if bit 3 is set
    Print "Bit 3 is set"
else
    Print "Bit 3 is cleared"
endif

vOut = vIn | (2^3)              // Set bit 3
Printf "Set bit 3: %d\r", vOut

vOut = vIn & ~(2^3)            // Clear bit 3
Printf "Clear bit 3: %d\r", vOut

vOut = vIn * (2^3)             // Shift three bits left
Printf "Shift three bits left: %d\r", vOut

vOut = vIn / (2^3)             // Shift three bits right
Printf "Shift three bits right: %d\r", vOut
End
```

For a simple demonstration, try this function passing 1 as the parameter.

In this example,  $2^3$  evaluates to 8 which is a value with only bit 3 set. In a real application, it is a good idea to define a constant that indicates the significance of the bit. These two statements are equivalent:

```
Constant kSystemReadyMask 8      // 8 specified in decimal notation
Constant kSystemReadyMask 0x08   // 8 specified in hexadecimal notation
```

## Switch Statements

The switch construct can sometimes be used to simplify complicated flow control. It chooses one of several execution paths depending on a particular value.

Instead of a single form of switch statement, as is the case in C, Igor has two types: *switch* for numeric expressions and *strswitch* for string expressions. The basic syntax of these switch statements is as follows:

```
switch(<numeric expression>)    // numeric switch
    case <literal number or numeric constant>:
        <code>
        [break]
    case <literal number or numeric constant>:
        <code>
        [break]
    . . .
    [default:
        <code>]
endswitch

strswitch(<string expression>)  // string switch
    case <literal string or string constant>:
        <code>
        [break]
    case <literal string or string constant>:
        <code>
        [break]
    . . .
    [default:
        <code>]
endswitch
```

The switch numeric or string expression is evaluated and execution proceeds with the code following the matching case label. When none of the case labels match, execution will continue at the default label, if it is present, or otherwise the switch will exit with no action taken.

All of the case labels must be numeric or string constant expressions and they must all have unique values within the switch statement. The constant expressions can either be literal values or they must be declared using the constant and strconstant keywords for numeric and string switches, respectively. For more about constants, see **Constants** on page IV-40.

Execution proceeds within each case until a break statement is encountered or the endswitch is reached. The break statement explicitly exits the switch construct. Usually, you will put a break statement at the end of each case. If you omit the break statement, execution continues with the next case label. Do this when you want to execute a single action for more than one switch value.

The following examples illustrate how switch constructs can be used in Igor:

```
Constant kThree=3
```

```
StrConstant ksHow="how"
```

```
Function NumericSwitch(a)
```

```
Variable a
```

```
switch(a) // numeric switch
```

```
case 1:
```

```
print "a is 1"
```

```
break
```

```
case 2:
```

```
print "a is 2"
```

```
break
```

```
case kThree:
```

```
case 4:
```

```
print "a is 3 or 4"
```

```
break
```

```
default:
```

```
print "a is none of those"
```

```
break
```

```
endswitch
```

```
End
```

```
Function StringSwitch(a)
```

```
String a
```

```
strswitch(a) // string switch
```

```
case "hello":
```

```
print "a is hello"
```

```
break
```

```
case ksHow:
```

```
print "a is how"
```

```
break
```

```
case "are":
```

```
case "you":
```

```
print "a is are or you"
```

```
break
```

```
default:
```

```
print "a is none of those"
```

```
break
```

```
endswitch
```

```
End
```

### Loops

Igor implements two basic types of looping structures: do-while and for loops. The do-while loop iterates through the loop code and tests an exit condition at the end of each iteration. The for loop is more complex; the beginning of a for loop includes expressions for initializing and updating variables as well as testing the loop's exit condition at the start of each iteration.

#### Do-While Loop

The form of the do-while loop structure is:

```
do
    <loop body>
while(<expression>)
```

This loop runs until the expression evaluates to zero or until a break statement is executed.

This example will always execute the body of the loop at least once, like the do-while loop in C.

```
Function Test(lim)
    Variable lim          // We use this parameter as the loop limit.

    Variable sum=0
    Variable i=0         // We use i as the loop variable.
    do
        sum += i        // This is the body; equivalent to sum=sum+i.
        i += 1          // Increment the loop variable.
    while(i < lim)
    return sum
End
```

#### Nested Do-While Loops

A nested loop is a loop within a loop. Here is an example:

```
Function NestedLoopTest(numOuterLoops, numInnerLoops)
    Variable numOuterLoops, numInnerLoops

    Variable i, j

    i = 0
    do
        j = 0
        do
            <inner loop body>
            j += 1
            while (j < numInnerLoops)
            i += 1
        while (i < numOuterLoops)
    End
```

#### While Loop

This fragment will execute the body of the loop zero or more times, like the while loop in C.

```
do
    if (i > lim)
        break          // This breaks out of the loop.
    endif
    <loop body>
    i += 1
while(1)              // This would loop forever without the break.
...                  // Execution continues here after the break.
```

In this example, the loop increment is 1 but it can be any value.

## For Loop

The basic syntax of a for loop is:

```
for(<initialize>;<continue test>;<update>)
  <loop body>
endfor
```

Here is a simple example:

```
Function Example1()
  Variable i

  for(i=0;i<5;i+=1)
    print i
  endfor
End
```

The beginning of a for loop consists of three semicolon-separated expressions. The first is usually an assignment statement that initializes one or more variables. The second is a conditional expression used to determine if the loop should be terminated — if true, nonzero, the loop is executed; if false, zero, the loop terminates. The third expression usually updates one or more loop variables.

When a for loop executes, the initialization expression is evaluated only once at the beginning. Then, for each iteration of the loop, the continuation test is evaluated at the start of every iteration, terminating the loop if needed. The third expression is evaluated at the end of the iteration and usually increments the loop variable.

All three expressions in a for statement are optional and can be omitted independent of the others; only the two semicolons are required. The expressions can consist of multiple assignments, which must be separated by commas.

In addition to the test expression, for loops may also be terminated by break or return statements within the body of the loop. A continue statement executed within the loop skips the remaining body code and execution continues with the loop's update expression.

Here is a more complex example:

```
Function Example2()
  Variable i,j

  for(i=0,j=10; ;i+=1,j*=2)
    if( i==2 )
      continue
    endif
    Print i,j
    if( i==5 )
      break
    endif
  endfor
End
```

## Break Statement

A break statement terminates execution of do-while loops, for loops, and switch statements. The break statement continues execution with the statement after the *enclosing* loop's while, endfor, or endswitch statement. A nested do-while loop example demonstrates this:

```
...
Variable i=0, j

do // Starts outer loop.
  if (i > numOuterLoops)
    break // Break #1, exits from outer loop.
  endif
```

```
j = 0
do           // Start inner loop.
    if (j > numInnerLoops)
        break // Break #2, exits from inner loop only.
    endif
    j += 1
while (1)   // Ends inner loop.
...        // Execution continues here from break #2.
i += 1
while (1)   // Ends outer loop.
...        // Execution continues here from break #1.
```

### Continue Statement

The continue statement can be used in do-while and for loops to short-circuit the loop and return execution back to the top of the loop. When Igor encounters a continue statement during execution, none of the code following the continue statement is executed during that iteration.

## Flow Control for Aborts

Igor Pro includes a specialized flow control construct and keywords that you can use to test for and respond to abort conditions. The `AbortOnRTE` and `AbortOnValue` keywords can be used to trigger aborts, and the `try-catch-endtry` construct can be used to control program execution when aborts occur. These are advanced techniques. If you are just starting with Igor programming, you may want to skip this section and come back to it later.

### AbortOnRTE Keyword

The `AbortOnRTE` keyword can be used to raise an abort whenever a runtime error occurs. Use `AbortOnRTE` immediately after (on the same line preceded by a semicolon) a command or on a line following a sequence of commands for which you wish to catch a runtime error.

See `AbortOnRTE` on page V-16 for further details. For a usage example see **try-catch-endtry Example** on page IV-39.

### AbortOnValue Keyword

The `AbortOnValue` keyword can be used to abort function execution when a specified abort condition is satisfied. When `AbortOnValue` triggers an abort, it can also return a numeric abort code that you can use to characterize the cause.

`AbortOnValue` is a low overhead, short-hand replacement for an

```
if (abortTest)
    Abort
endif
```

code block that you would normally use in a procedure to test for an abort condition.

See `AbortOnValue` on page V-16 for further details. For a usage example see **try-catch-endtry Example** on page IV-39.

### try-catch-endtry Flow Control

The `try-catch-endtry` flow control construct has the following syntax:

```
try
    <possible abort code>
catch
    <code to handle abort>
endtry
```

The try-catch-entry flow control construct can be used to catch and respond to abort conditions in user functions. Code within the try-catch area tests for abort conditions and when the first abort condition is satisfied, execution will immediately jump to code within the catch-endtry area where execution proceeds with code to handle the abort condition. Normal flow (no aborts) will skip past all code within the catch-endtry area.

When an abort occurs within the try-catch area, the construct returns a numeric code in the V\_AbortCode variable, which provides information about the cause of the abort.

See **try-catch-endtry** on page V-713 for further details.

### try-catch-endtry Example

The following example demonstrates how abort flow control may be used. Copy the code to your Procedure window and execute the foo function with 0 to 6 as input parameters.

```
Function foo(a)
  Variable a

  print "A"
  try
    print "B1"
    AbortOnValue a==1 || a==2,33
    print "B2"
    bar(a)
    print "B3"
    try
      print "C1"
      if( a==4 || a==5 )
        Make $" "; AbortOnRTE
      endif
      Print "C2"
    catch
      Print "D1"
      // will be runtime error so pass along to outer catch
      AbortOnValue a==5, V_AbortCode
      Print "D2"
    endtry
    Print "B4"
    if( a==6 )
      do
        while(1)
        endif
      Print "B5"
    catch
      print "Abort code= ", V_AbortCode
      if( V_AbortCode == -4 )
        Print "Runtime error= ", GetRTEError(1)
      endif
      if( a==1 )
        abort "Aborting again"
      endif
      Print "E"
    endtry
    print "F"
  End

Function bar(b)
  Variable b

  Print "Bar A"
  AbortOnValue b==3,99
```

```
    Print "Bar B"  
End
```

### Constants

You can define named numeric and string constants in Igor procedure files and use them in the body of user-defined functions.

Constants are defined in procedure files using following syntax:

```
Constant <name1> = <literal number> [, <name2> = <literal number>]  
StrConstant <name1> = <literal string> [, <name2> = <literal string>]
```

For example:

```
Constant kIgorStartYear=1989,kIgorEndYear=2020  
StrConstant ksPlatformMac="Macintosh",ksPlatformWin="Windows"
```

```
Function Test1()  
    Variable v1 = kIgorStartYear  
    String s1 = ksPlatformMac  
    Print v1, s1  
End
```

Constants declared like this are public and can be used in any function in any procedure file. A typical use would be to define constants in a utility procedure file that could be used from other procedure files as parameters to the utility routines. Be sure to use precise names to avoid conflicts with public constants declared in other procedure files.

If you are defining constants for use in a single procedure file, for example to improve readability or make the procedures more maintainable, you should use the `static` keyword (see **Static** on page V-594 for details) to limit the scope to the given procedure file.

```
static Constant kStart=1989,kEnd=2020  
static StrConstant ksMac="Macintosh",ksWin="Windows"
```

We suggest that you use the “k” prefix for numeric constants and the “ks” prefix for string constants. This makes it immediately clear that a particular keyword is a constant.

Names for numeric and string constants are allowed to conflict with all other names. Duplicate constants of a given type are not allowed (except static in different files and when used with the override keyword). The only true conflict is with variable names and with certain built-in functions that do not take parameters such as pi. Variable names (including local variable names, waves, NVARs, and SVARs) override constants, but constants override functions such as pi.

### Pragmas

A pragma is a statement in a procedure file that sets a compiler mode or passes other information from the programmer to Igor. The form of a pragma statement is:

```
#pragma keyword [= parameter]
```

The pragma statement must be flush against the left margin of the procedure window, with no indentation.

Currently, Igor supports the following pragmas:

```
#pragma rtGlobals = value  
#pragma version = versionNumber  
#pragma IgorVersion = versionNumber  
#pragma ModuleName = name  
#pragma IndependentModule = name
```

The effect of a pragma statement lasts until the end of the procedure file that contains it.

## The rtGlobals Pragma

The rtGlobals pragma controls aspects of the Igor compiler and runtime error checking in user-defined functions.

Prior to Igor Pro 3, to access a global (wave or variable) from a user-defined function, the global had to already exist. Igor Pro 3 introduced "runtime lookup of globals" under which the Igor compiler did not require globals to exist at compile time but rather connected references, declared with WAVE, NVAR and SVAR statements, to globals at runtime. Igor Pro 6.20 introduced stricter compilation of wave references and runtime checking of wave index bounds.

You enable and disable these behaviors using an rtGlobals pragma. For example:

```
#pragma rtGlobals = 3    // Strict wave reference mode, runtime bounds checking
```

A given rtGlobals pragma governs just the procedure file in which it appears. The pragma must be flush left in the procedure file and is typically put at the top of the file.

The rtGlobals pragma is defined as follows:

#pragma rtGlobals=0	Specifies the old, pre-Igor Pro 3 behavior. This is obsolete and should not be used.
#pragma rtGlobals=1	Turns on runtime lookup of globals. In Igor Pro 5 or later, this is the default setting if there is no rtGlobals pragma in a given procedure file.
#pragma rtGlobals=2	Forces old experiments out of compatibility mode. This is superceded by rtGlobals=3. It is described under <b>Legacy Code Issues</b> on page IV-90.
#pragma rtGlobals=3	Turns on runtime lookup of globals, strict wave references and runtime checking of wave index bounds. Requires Igor Pro 6.2 or later.

If your procedures will run only with Igor Pro 6.20 or later, rtGlobals=3 is recommended. Otherwise rtGlobals=1 is recommended.

If your procedures must run with old Igor versions but you want to use rtGlobals=3 when possible, you can do this:

```
#if IgorVersion() >= 6.20
#pragma rtGlobals = 3    // Strict wave reference mode, runtime bounds checking
#endif
```

Since rtGlobals=1 is the default, this will use rtGlobals=1 for Igor Pro 6.02A through 6.12 and will use rtGlobals=3 for Igor Pro 6.20 or later. It will not work for versions prior to 6.02A because they don't support conditional compilation (#if).

Under strict wave references (rtGlobals=3), you must create a wave reference for any use of a wave. Without strict wave references (rtGlobals=1), you do not need to create a wave reference unless the wave is used in an assignment statement. For example:

```
Function Test()
    jack = 0          // Error under rtGlobals=1 and under rtGlobals=3
    Display jack     // OK under rtGlobals=1, error under rtGlobals=3

    Wave jack        // jack is now a wave reference rather than a bare name
    Display jack     // OK under rtGlobals=1 and under rtGlobals=3
End
```

Even with rtGlobals=3, this compiles without error:

```
Function Test()
    // Make creates an automatic wave reference when used with a simple name
    Make jack
```

## Chapter IV-3 — User-Defined Functions

---

```
    Display jack    // OK under rtGlobals=1 and rtGlobals=3
End
```

See **Automatic Creation of WAVE References** on page IV-56 for details.

Under runtime wave index checking (rtGlobals=3), Igor reports an error if a wave index is out-of-bounds:

```
Function Test()
    Make/O/N=5 jack = 0 // Creates automatic wave reference

    jack[4] = 123      // OK
    jack[5] = 234      // Runtime error under rtGlobals=3.
                       // Clipped under rtGlobals=1.

    Variable index = 5
    jack[index] = 234  // Runtime error under rtGlobals=3.
                       // Clipped under rtGlobals=1.

    // Create and use a dimension label for point 4 of jack
    SetDimLabel 0,4,four,jack
    jack[%four] = 234 // OK

    // Use a non-existent dimension label.
    jack[%three] = 345 // Runtime error under rtGlobals=3.
                       // Clipped under rtGlobals=1.
    // Under rtGlobals=1, this statement writes to point 0 of jack.
End
```

**NOTE:** All Igor documentation assumes that rtGlobals=1 is in effect unless otherwise stated.

See also: **Runtime Lookup of Globals** on page IV-50

**Automatic Creation of WAVE References** on page IV-56

**Automatic Creation of NVAR and SVAR References** on page IV-55

**Legacy Code Issues** on page IV-90

### The version Pragma

The version pragma sets the version of the procedure file. It is optional and is of interest mostly if you are the developer of a package used by a widespread group of users.

For details on the version pragma, see **Procedure File Version Information** on page IV-145.

### The IgorVersion Pragma

The IgorVersion pragma is also optional and of interest to developers of packages. It gives you a way to prevent procedures from running, or at least generating compilation errors that cannot be fixed, under versions of Igor Pro older than the specified version number.

For example, the statement:

```
#pragma IgorVersion = 4.0
```

requires Igor Pro 4.0 or later for the procedure file. This will be helpful if you share your code with others because you can ensure that your procedures will only run with versions of Igor Pro that fully support all of the new or updated features you may have used. This Igor version check was added with Igor Pro 4.0, and it will not work with previous versions.

### The ModuleName Pragma

The ModuleName pragma gives you the ability to use static functions and Proc Pictures in a global context, such as in the action procedure of a control or on the Command Line. Using this pragma entails a two step process: define a name for the procedure file, and then use a special syntax to access objects in the named procedure file.

To define a module name for a procedure file use the format:

```
#pragma ModuleName= name
```

This statement associates the specified module name with the procedure file in which the statement appears.

You can then use objects from the named procedure file by preceding the object name with the name of the module and the # character. For example:

```
#pragma ModuleName= myGreatProcedure
Static Function foo(a)
  Variable a

  return a+100
End
```

Then on the command line you can execute:

```
Print myGreatProcedure#foo(3)
```

You should make sure the name is unlikely to clash with other names you or others might use. WaveMetrics will use names with a `WM_` prefix, so you should avoid such names. The `ModuleName` pragma will not work with versions previous to Igor Pro 5.0.

For further discussion see **Regular Modules** on page IV-212.

## The IndependentModule Pragma

The `IndependentModule` pragma is a way for you to designate groups of one or more procedure files that are compiled and linked separately. Once compiled and linked, the code remains in place and is usable even though other procedures may fail to compile. This allows functioning control panels and menus to continue to work regardless of user programming errors.

A file is designated as an independent module using

```
#pragma IndependentModule=imName
```

This is similar to `#pragma ModuleName=modName` (see **The ModuleName Pragma** on page IV-42) and, just as in the case of calling static functions in a procedure with `#pragma ModuleName`, calling nonstatic function in an `IndependentModule` from outside the module requires the use of `imName#functionName()` syntax.

For further discussion see **Independent Modules** on page IV-214.

## Unknown Pragmas

Starting with Igor Pro version 3.04, Igor ignores pragmas that it does not know about. This allows newer versions of Igor to use new pragmas while older versions ignore them. The downside of this change is that, if you misspell a pragma keyword, Igor will not warn you about it.

## Proc Pictures

Proc Pictures are binary PNG or JPEG images encoded as printable ASCII procedures in procedure files. They are intended for programmers who need images as part of the user interface for a procedure package. They can be used with the **DrawPICT** operation (see page V-128) and with the **Picture** keyword (see page V-480) in certain controls.

The syntax for defining and using a Proc Picture is illustrated in the following example:

```
// PNG: width= 56, height= 44
Picture myPictName
ASCII85Begin
M,6r;%14!\!!!!.8Ou6I!!!!Y!!!!M#Qau+!5G;q_uKc;&TgHDFAm*iFE_/6AH5;7DfQssEc39jTBQ
=U!7FG,5u`*!m?g0PK.mR"U!k63rtBW)]$T)Q*!=Sa1TCDV*V+l:Lh^NW!fu1>;(.<VU1bs4L8&@Q_
```

## Chapter IV-3 — User-Defined Functions

---

```
<4e(%^F50:Jg6);j!CQdUA[dh6] %[OkHSC,ht+Q7ZO#.6U,IgfsZ!R1g':oO_iLF.GQ@RF[/*G98D
bjE.g?NCte(pX-($m^\_FhhfL`D9uO6Qi5c[r4849Fc7+*)*O[tY(6<rkm^)/KLIc]VdDEbF-n5&Am
2^hbTu:U#8ies_W<LGkp_LEU1bs4L8&?fqRJ[h#sVSSz8OZBBY!QNJ
ASCII85End
End

Function test()
  NewPanel
  DrawPict 0,0,1,1,ProcGlobal#myPictName
End
```

The ASCII text in the `myPictName` procedure between the `ASCII85Begin` and `ASCII85End` is similar to output from the Unix `base64` command or, on Macintosh, StuffIt's binary to ASCII operation. (You must remove any extra header and trailer information if you use either of these utilities to create a Proc Picture.)

You can create Proc Pictures in Igor Pro from normal, global pictures using the Picture dialog (see **Pictures** on page III-421). Select a picture in the dialog and click the Copy Proc Picture button to place the text on the Clipboard and then paste it in your procedure file. If the existing picture is not a JPEG or PNG, it is converted to PNG.

Proc Pictures can be either global or local in scope. Global pictures can be used in all experiment procedure files; local pictures can be used only within the procedure file where they are defined. Proc Pictures are global by default and the picture name must be unique for all procedure files in an experiment. Proc Pictures can be made local in scope by declaring them **Static** (see **Static** on page V-594).

When accessing a Proc Picture from `DrawPict`, the picture name must be preceded by either the `ProcGlobal` keyword or the procedure module name, and the two names joined together with `#`. This naming convention is necessary to avoid potential conflicts with any existing experiment global pictures defined via the Pictures dialog.

For a global Proc Picture, you must use the `ProcGlobal` keyword as the prefix:

```
ProcGlobal#gProcPictName
```

For a static Proc Picture, you must use the module name defined in the procedure file by the `#pragma ModuleName = modName` statement (see **ModuleName** on page V-424) as the prefix:

```
modName#ProcPictName
```

## How Parameters Work

There are two ways of passing parameters from a routine to a subroutine: **pass-by-value** and **pass-by-reference**. "Pass-by-value" means that the routine passes the *value* of an expression to the subroutine. "Pass-by-reference" means that the routine passes *access to a variable* to the subroutine. The important difference is that, in pass-by-reference, the subroutine can change the original variable in the calling routine.

The C and Pascal languages allow programmers to use either of these techniques, at their option while FORTRAN uses pass-by-reference only. Like C, Igor also allows either method for numeric and string variables.

### Example of Pass-By-Value

```
Function Routine()
  Variable v = 4321
  String s = "Hello"

  Subroutine(v, s)
End

Function Subroutine(v, s)
  Variable v
  String s

  Print v, s
```

```

// These lines have NO EFFECT on the calling routine.
v = 1234
s = "Goodbye"
End

```

Note that *v* and *s* are *local variables* in Routine. In Subroutine, they are *parameters* which act very much like local variables. The names “*v*” and “*s*” are local to the respective functions. The *v* in Subroutine is not the same variable as the *v* in Routine although it initially has the same value.

The last two lines of Subroutine set the value of the local variables *v* and *s*. They have no effect on the value of the variables *v* and *s* in the calling Routine. What is passed to Subroutine is the numeric value 4321 and the string value “Hello”.

## Pass-By-Reference

You can specify that a parameter to a function is to be passed by reference rather than by value. In this way, the function called can change the value of the parameter and update it in the calling function. This is much like using pointers to arguments in C. This technique is needed and appropriate only when you need to return more than one value from a function.

The variable or string being passed must be a local variable and can not be a global variable. To designate a variable or string parameter for pass-by-reference, simply prepend an ampersand symbol (&) before the name in the parameter declaration:

```

Function Subroutine(num1,num2,str1)
  Variable &num1, num2
  String &str1

  num1= 12+num2
  str1= "The number is"
End

```

and then call the function with the name of a local variable in the reference slot:

```

Function Routine()
  Variable num= 1000
  String str= "hello"
  Subroutine(num,2,str)
  print str, num
End

```

When executed, Routine prints “The number is 14” rather than “hello 1000”, which would be the case if pass-by-reference were not used.

A pass-by-reference parameter can be passed to another function that expects a reference:

```

Function SubSubroutine(b)
  Variable &b
  b= 123
End

Function Subroutine(a)
  Variable &a
  SubSubroutine(a)
End

Function Routine()
  Variable num
  Subroutine(num)
  print num
End

```

**Note:** Functions with pass-by-reference parameters can only be called from other functions — not from the command line.

## Chapter IV-3 — User-Defined Functions

---

### How Waves Are Passed

Here is an example of a function “passing a wave” to a subroutine.

```
Function Routine()
    Make/O wave0 = x
    Subroutine(wave0)
End

Function Subroutine(w)
    WAVE w

    w = 1234    // This line DOES AFFECT the wave referred to by w.
End
```

We are really not passing a wave to Subroutine, but rather we are passing a reference to a wave. The parameter *w* is the wave reference.

Waves are global objects that exist independent of functions. The subroutine can use the wave reference to modify the contents of the wave. Using the terminology of “pass-by-value” and “pass-by-reference”, the wave reference is passed by value, but this has the effect of “passing the wave” by reference.

### Using Optional Parameters

Following is an example of a function with two optional input parameters:

```
Function opParamTest(a,b, [c,d])
    Variable a,b,c,d          // c and d are optional parameters

    // Determine if the optional parameters were supplied
    if( ParamIsDefault(c) && ParamIsDefault(d) )
        Print "Missing optional parameters c and d."
    endif

    Print a,b,c,d
End
```

Executing on the Command Line with none of the optional inputs:

```
•opParamTest(8,6)
  Missing optional parameters c and d.
  8 6 0 0
```

Executing with an optional parameter as an input:

```
•opParamTest(8,6, c=66)
  8 6 66 0
```

Note that the optional parameter is explicitly defined in the function call using the *ParamName=Value* syntax. All optional parameters must come after any required function parameters.

## Local Versus Global Variables

Numeric and string variables can be **local** or **global**. “Local” means that the variable can be accessed only from within the procedure in which it was created. “Global” means that the variable can be accessed from the command line or from within any procedure. The following table shows the characteristics of local and global variables:

Local Variables	Global Variables
Are part of a procedure.	Are part of an Igor experiment.
Created using Variable or String within a procedure.	Created using Variable or String or Variable/G or String/G from the command line or within a procedure.

---

Local Variables	Global Variables
Used to store temporary results while the procedure is executing.	Used to store values that are saved from one procedure invocation to the next or to store data that is accessed by many procedures.
Cease to exist when the procedure ends.	Exist until you use KillVariables, KillStrings, or KillDataFolder.
Can be accessed only from within the procedure in which they were created.	Can be accessed from the command line or from within any procedure.

Local variables are private to the function in which they are declared and vanish when the function exits. Global variables are public and persistent — they exist until you explicitly delete them.

If you write a function whose main purpose is to display a dialog to solicit input from the user, you may want to store the user's choices in global variables and use them to initialize the dialog the next time the user invokes it. This is an example of saving values from one invocation of a function to the next.

If you write a set of functions that loads, displays and analyzes experimentally acquired data, you may want to use global variables to store values that describe the conditions under which the data was acquired and to store the results from the analyses. These are examples of data accessed by many procedures.

## Local Variables Used by Igor Operations

When invoked from a function, a number of Igor's operations return results via local variables. For example, the WaveStats operation creates a number of local variables with names such as V\_avg, V\_sigma, etc. The following function illustrates this point.

```
Function PrintWaveAverage(w)
    WAVE w

    WaveStats/Q w
    Print V_avg
End
```

When the Igor compiler compiles the WaveStats operation, it creates various local variables, V\_avg among them. When the WaveStats operation runs, it stores results in these local variables.

In addition to creating local variables, a few operations, such as CurveFit and FuncFit, check for the existence of specific local variables to provide optional behavior. For example:

```
Function ExpFitWithMaxIterations(w, maxIterations)
    WAVE w
    Variable maxIterations

    Variable V_FitMaxIters = maxIterations

    CurveFit exp w
End
```

The CurveFit operation looks for a local variable named V\_FitMaxIters, which sets the maximum number of iterations before the operation gives up.

The documentation for each operation lists the special variables that it creates or looks for.

## Converting a String into a Reference Using \$

The \$ operator converts a string expression into an object reference. The referenced object is usually a wave but can also be a global numeric or global string variable, a window, a symbolic path or a function. This is a common and important technique.

## Chapter IV-3 — User-Defined Functions

---

We often use a *string* to pass the *name* of a wave to a procedure or to algorithmically generate the name of a wave. Then we use the \$ operator to convert the string into a wave reference so that we can operate on the wave.

The following trivial example shows why we need to use the \$ operator:

```
Function DisplayXY(xWaveNameStr, yWaveNameStr)
    String xWaveNameStr, yWaveNameStr
```

```
    Display $yWaveNameStr vs $xWaveNameStr
End
```

Here we use \$ to convert the string parameters into wave references. The function will display the wave whose name is stored in the yWaveNameStr string versus the wave whose name is stored in the xWaveNameStr string.

If we omitted the \$ operators, we would have

```
Display yWaveNameStr vs xWaveNameStr
```

This would result in an error because the Display operation would look for a wave named yWaveName and would not find it.

As shown in the following example, \$ can create references to global numeric and string variables as well as to waves.

```
Function Test(vStr, sStr, wStr)
    String vStr, sStr, wStr

    NVAR v = $vStr          // v is local name for global numeric var
    v += 1
    SVAR s = $sStr          // s is local name for global string var
    s += "Hello"
    WAVE w = $wStr          // w is local name for global wave
    w += 1
End
```

```
Variable/G gVar = 0; String/G gStr = ""; Make/O/N=5 gWave = p
Test("gVar", "gStr", "gWave")
```

The NVAR, SVAR and WAVE references are necessary in functions so that the compiler can identify the kind of object. This is explained under **Accessing Global Variables and Waves** on page IV-50.

### Using \$ to Refer to a Window

A number of Igor operations modify or create windows, and optionally take the name of a window. You need to use a string variable if the window name is not determined until run time but must convert the string into a name using \$.

For instance, this function creates a graph using a name specified by the calling function:

```
Function DisplayXY(xWaveNameStr, yWaveNameStr, graphNameStr)
    String xWaveNameStr, yWaveNameStr
    String graphNameStr          // Contains name to use for the new graph

    Display /N=$graphNameStr $yWaveNameStr vs $xWaveNameStr
    Cursor /W=$graphNameStr A, $yWaveNameStr, 0
End
```

The \$ operator in /N=\$graphNameStr converts the contents of the string graphNameStr into a graph name as required by the Display operation /N flag. If you forget \$, the command would be:

```
Display /N=graphNameStr $yWaveNameStr vs $xWaveNameStr
```

This would create a graph literally named graphNameStr.

After creating the graph, this example uses \$graphNameStr again to specify the target of the Cursor operation.

## Using \$ In a Data Folder Path

\$ can also be used to convert a string to a name in a data folder path. This is used when one of many data folders must be selected algorithmically.

Assume you have a string variable named dfName that tells you in which data folder a wave should be created. You can write:

```
Make/O root:$(dfName):wave0
```

The parentheses are necessary because the \$ operator has low precedence, so Igor would interpret this:

```
Make/O root:$dfName:wave0 // ERROR
```

to mean:

```
Make/O root:$(dfName:wave0) // ERROR
```

To avoid this, you must use parentheses like this:

```
Make/O root:$(dfName):wave0 // OK
```

## Compile Time Versus Runtime

Because Igor user-defined functions are compiled, you need to be aware of the difference between “compile time” and “runtime”.

Compile time is when Igor analyzes the text of all functions and produces low-level instructions that can be executed quickly later. This happens when you modify a procedure window and then:

- Choose Compile from the Macros menu.
- Click the Compile button at the bottom of a procedure window.
- Activate a nonprocedure window.

Runtime is when Igor actually executes a function’s low-level instructions. This happens when:

- You invoke the function from the command line.
- The function is invoked from another procedure.
- Igor updates a dependency which calls the function.
- You use a button or other control that calls the function.

Conditions that exist at compile time are different from those at runtime. For example, a function can reference a global variable. The global does not need to exist at compile time, but it does need to exist at runtime. This issue is discussed in detail in the following sections.

Here is another example of the distinction between compile time and runtime:

```
Function Example(w)
  WAVE w

  w= sin(x)
  FFT w
  w= r2polar(w)
End
```

The declaration “WAVE w” specifies that w is expected to be a real wave. This is correct until the FFT executes and thus the first wave assignment produces the correct result. After the FFT is executed at runtime, however, the wave becomes complex. The Igor compiler does not know this and so it will compile the second wave assignment on the assumption that w is real. A compile-time error will be generated com-

## Chapter IV-3 — User-Defined Functions

---

plaining that `r2polar` is not available for this number type — i.e., real. To provide Igor with the information that the wave is complex after the FFT you need to rewrite the function like this:

```
Function Example(w)
    WAVE w

    w= sin(x)
    FFT w
    WAVE/C wc= w
    wc= r2polar(wc)
End
```

A statement like “`WAVE/C wc= w`” has the compile-time behavior of creating a symbol, `wc`, and specifying that it refers to a complex wave. It has the runtime behavior of making `wc` refer to a specific wave. The runtime behavior can not occur at compile time because the wave may not exist at compile time.

## Accessing Global Variables and Waves

Global numeric variables, global string variables and waves can be referenced from any function. A function can refer to a global that does not exist at compile-time. For the Igor compiler to know what type of global you are trying to reference, you need to declare references to globals.

Consider the following function:

```
Function BadExample()
    gStr1 = "The answer is:"
    gNum1 = 1.234
    wave0 = 0
End
```

The compiler can not compile this because it doesn't know what `gStr1`, `gNum1` and `wave0` are. We need to specify that they are a global string variable, a global numeric variable and a wave, respectively:

```
Function GoodExample1()
    SVAR gStr1 = root:gStr1
    NVAR gNum1 = root:gNum1
    WAVE wave0 = root:wave0

    gStr1 = "The answer is:"
    gNum1 = 1.234
    wave0 = 0
End
```

The `SVAR` statement specifies two important things for the compiler: first, that `gStr1` is a global string variable; second, that `gStr1` refers to a global string variable named `gStr1` in the root data folder. Similarly, the `NVAR` statement identifies `gNum1` and the `WAVE` statement identifies `wave0`. With this knowledge, the compiler can compile the function.

The technique illustrated here is called “runtime lookup of globals” because the compiler compiles code that associates the symbols `gStr1`, `gNum1` and `wave0` with specific global variables at runtime.

## Runtime Lookup of Globals

The syntax for runtime lookup of globals is:

```
NVAR <local name1>[= <path to var1>][, <loc name2>[= <path to var2>]]...
SVAR <local name1>[= <path to str1>][, <loc name2>[= <path to str2>]]...
WAVE <local name1>[= <path to wave1>][, <loc name2>[= <path to wave2>]]...
```

`NVAR` and `SVAR` create a reference to a global numeric or string variable and `WAVE` creates a reference to a wave. At compile time, these statements identify the referenced objects. At runtime, the connection is made between the local name and the actual object. Consequently, the object must exist when these statements are executed.

<local name> is the name by which the global variable, string or wave is to be known within the user function. It does not need to be the same as the name of the global variable. The example function could be rewritten as follows:

```
Function GoodExample2()
  SVAR str1 = root:gStr1      // str1 is the local name.
  NVAR num1 = root:gNum1     // num1 is the local name.
  WAVE w = root:wave0       // w is the local name.

  str1 = "The answer is:"
  num1 = 1.234
  w = 0
End
```

If you use a local name that is the same as the global name, and if you want to refer to a global in the current data folder, you can omit the <path to ...> part of the declaration:

```
Function GoodExample3()
  SVAR gStr1                // Refers to gStr1 in current data folder.
  NVAR gNum1                // Refers to gNum1 in current data folder.
  WAVE wave0                // Refers to wave0 in current data folder.

  gStr1 = "The answer is:"
  gNum1 = 1.234
  wave0 = 0
End
```

GoodExample3 accesses globals in the current data folder while GoodExample2 accesses globals in a specific data folder.

If you use <path to ...>, it may be a simple name (gStr1) or it may include a full or partial path to the name.

The following are valid examples, referencing a global numeric variable named gAvg:

```
NVAR gAvg= gAvg
NVAR avg= gAvg
NVAR gAvg
NVAR avg= root:Packages:MyPackage:gAvg
NVAR avg= :SubDataFolder:gAvg
NVAR avg= $"gAvg"
NVAR avg= $("g"+ "Avg")
NVAR avg= ::$"gAvg"
```

As illustrated above, the local name can be the same as the name of the global object and the lookup expression can be either a literal name or can be computed at runtime using \$<string expression>.

In some cases, it may be convenient to create global variables in a temporary data folder as a way of passing results back to the calling routine. For example:

```
Function MyWaveStats(inputWave)
  WAVE inputWave

  NewDataFolder/O/S tmpMyWaveStatsDF

  WaveStats/Q inputWave

  Variable/G gNumPoints = V_npnts
  Variable/G gAvg = V_avg
  Variable/G gSdev = V_sdev
End

Function Test()
  Make/O testwave= gnoise(1)

  MyWaveStats(testwave)      // Create temp output data folder.
```

## Chapter IV-3 — User-Defined Functions

---

```
NVAR gNumPoints,gAvg,gSdev // Create references to globals.
Printf "Points: %g; Avg: %g; SDev: %g\r", gNumPoints,gAvg,gSdev
KillDataFolder :           // Kill temp output data folder.
End
```

Note that the NVAR statement must appear after the MyWaveStats call because NVAR associates the local names with existing global variables at runtime.

### Put WAVE Declaration After Wave Is Created

A wave declaration serves two purposes. At compile time, it tells Igor the local name and type of the wave. At runtime, it connects the local name to the wave. In order for the runtime purpose to work, you must put wave declaration after the wave is created.

```
Function BadExample()
  String path = "root:Packages:MyPackage:wave0"
  Wave w = $path           // WRONG: Wave does not yet exist.
  Make $path
  w = p                    // w is not connected to any wave.
End
```

```
Function GoodExample()
  String path = "root:Packages:MyPackage:wave0"
  Make $path
  Wave w = $path           // RIGHT
  w = p
End
```

Both of these functions will successfully compile. BadExample will fail at runtime because w is not associated with a wave, because the wave does not exist when the "Wave w = \$path" statement executes.

This rule also applies to NVAR and SVAR declarations.

### Runtime Lookup Failure

At runtime, it is possible that a NVAR, SVAR or WAVE statement may fail. For example, NVAR v1= var1 will fail if var1 does not exist in the current data folder when the statement is executed. You can use the NVAR\_Exists, SVAR\_Exists, and WaveExists functions to test if a given global reference is valid.

One cause for failure is putting a WAVE statement in the wrong place. For example:

```
Function BadExample()
  WAVE w = resultWave
  <Call a function that creates a wave named resultWave>
  Display w
End
```

This function will compile successfully but will fail at runtime. The reason is that the "WAVE w = resultWave" statement has the runtime behavior of associating the local name w with a particular wave. But that wave does not exist until the following statement is executed. The function should be rewritten as:

```
Function GoodExample()
  <Call a function that creates a wave named resultWave>
  WAVE w = resultWave
  Display w
End
```

### Runtime Lookup Failure and the Debugger

You can break whenever a runtime lookup fails using the symbolic debugger (described in Chapter IV-8, **Debugging**). It is a good idea to do this, because it lets you know about runtime lookup failures at the moment they occur.

Sometimes you may create a WAVE, NVAR or SVAR reference knowing that the referenced global may not exist at runtime. Here is a trivial example:

```
Function Test()
    WAVE w = testWave
    if (WaveExists(testWave))
        Printf "testWave had %d points.\r", numpnts(testWave)
    endif
End
```

If you enable the debugger's WAVE checking and if you execute the function when testWave does not exist, the debugger will break and flag that the WAVE reference failed. But you wrote the function to handle this situation, so the debugger break is not helpful in this case.

The solution is to rewrite the function using WAVE/Z instead of just WAVE. The /Z flag specifies that you know that the runtime lookup may fail and that you don't want to break if it does. You can use NVAR/Z and SVAR/Z in a similar fashion.

### Accessing Complex Global Variables and Waves

You must specify if a global numeric variable or a wave is complex using the /C flag:

```
NVAR/C gc1= gc1
WAVE/C gcw1= gcw1
```

### Accessing Text Waves

Text waves must be accessed using the /T flag:

```
WAVE/T tw= MyTextWave
```

### Accessing Global Variables and Waves Using Liberal Names

There are two ways to reference an Igor object: using a literal name or path or using a string variable. For example:

```
Wave w = root:MyDataFolder:MyWave // Using literal path
String path = "root:MyDataFolder:MyWave"
Wave w = $path // Using string variable
```

Things get more complicated when you use a liberal name rather than a standard name. A standard name starts with a letter and includes letters, digits and the underscore character. A liberal name includes other characters such as spaces or punctuation.

In general, you must quote liberal names using a single quote so that Igor can determine where the name starts and where it ends. For example:

```
Wave w = root:'My Data Folder':'My Wave' // Using literal path
String path = "root:'My Data Folder':'My Wave'"
Wave w = $path // Using string variable
```

However, there is an exception to the quoting requirement. The rule is:

You must quote a literal liberal name and you must quote a liberal path stored in a string variable but you must not quote a simple liberal liberal name stored in a string variable.

The following functions illustrate this rule:

```
// Literal liberal name must be quoted
Function DemoLiteralLiberalNames()
    NewDataFolder/O root:'My Data Folder'

    Make/O root:'My Data Folder':'My Wave' // Literal name must be quoted
```

## Chapter IV-3 — User-Defined Functions

---

```
SetDataFolder root:'My Data Folder' // Literal name must be quoted

Wave w = 'My Wave' // Literal name must be quoted
w = 123

SetDataFolder root:
End

// String liberal PATH must be quoted
Function DemoStringLiberalPaths()
String path = "root:'My Data Folder'"
NewDataFolder/O $path

path = "root:'My Data Folder':'My Wave'" // String path must be quoted
Make/O $path

Wave w = $path
w = 123

SetDataFolder root:
End

// String liberal NAME must NOT be quoted
Function DemoStringLiberalNames()
SetDataFolder root:

String dfName = "My Data Folder" // String name must NOT be quoted
NewDataFolder/O $dfName

String wName = "My Wave" // String name must NOT be quoted
Make/O root:$(dfName):$wName

Wave w = root:$(dfName):$wName // String name must NOT be quoted
w = 123

SetDataFolder root:
End
```

The last example illustrates another subtlety. This command would generate an error at compile time:

```
Make/O root:$dfName:$wName // ERROR
```

because Igor would interpret it as:

```
Make/O root:$(dfName:$wName) // ERROR
```

To avoid this, you must use parentheses like this:

```
Make/O root:$(dfName):$wName // OK
```

### Runtime Lookup Example

In this example, a function named Routine calls another function named Subroutine and needs to access a number of result values created by Subroutine. To make it easy to clean up the temporary result globals, Subroutine creates them in a new data folder. Routine uses the results created by Subroutine and then deletes the temporary data folder.

```
Function Subroutine(w)
WAVE w

NewDataFolder/O/S SubroutineResults // Results go here
```

```

WaveStats/Q w           // WaveStats creates local variables
Variable/G gAvg= V_avg // Return the V_avg result in global gAvg
Variable/G gMin= V_min
String/G gWName= NameOfWave(w)

SetDataFolder ::       // Back to original data folder
End

Function Routine()
  Make aWave= {1,2,3,4}
  Subroutine(aWave)
  NVAR theAvg= :SubroutineResults:gAvg // theAvg is local name
  NVAR theMin= :SubroutineResults:gMin
  SVAR theName= :SubroutineResults:gWName
  Print theAvg,theMin,theName
  KillDataFolder SubroutineResults // We are done with results
End

```

Note that the NVAR statement must appear *after* the procedure (Subroutine in this case) that creates the global variable. This is because NVAR has both a compile-time and a runtime behavior. At compile time, it creates a name that Igor can compile (theAvg in this case). At runtime, it actually looks up and creates a link to the global (variable gAvg stored in data folder SubroutineResults in this case).

Often a Function will need to access quite a large number of global variables stored in a data folder. In such cases, you can write more compact code using the ability of NVAR, SVAR or WAVE to access multiple objects in the current data folder as illustrated here:

```

Function Routine2()
  Make/O aWave= {1,2,3,4}
  Subroutine(aWave)

  String dfSav=GetDataFolder(1)
  SetDataFolder :SubroutineResults
  NVAR gAvg,gMin // Access two variables via one NVAR
  SVAR gWName
  SetDataFolder dfSav

  Print gAvg,gMin,gWName
  KillDataFolder SubroutineResults // We are done with results
End

```

### Automatic Creation of NVAR and SVAR References

The Igor compiler sometimes automatically creates NVAR and SVAR references. For example:

```

Function Example1()
  Variable/G gVar1
  gVar1= 1

  String/G gStr1
  gStr1= "hello"
End

```

In this example we did not use NVAR or SVAR references and yet we were still able to compile assignment statements referring to global variables which will not exist until runtime. This is a feature of Variable/G and String/G that automatically create local references for simple object names.

Simple object names are names which are known at compile time for objects which will be created in the current data folder at runtime. Variable/G and String/G do not create references if you use \$<name>, a partial data folder path or a full data folder path to specify the object.

### Wave References

A wave reference is a lightweight object used in user-defined functions to specify the wave of interest in assignment statements and other commands. You can think of a wave reference as an identification number that Igor uses to identify a particular wave.

Wave reference variables hold wave references. They can be created as local variables, passed as parameters and returned as function results.

Here is a simple example:

```
Function Test(wIn)
    Wave wIn          // Reference to the input wave received as parameter

    String newName = NameOfWave(wIn) + "_out" // Compute output wave name

    Duplicate/O wIn, $newName                // Create output wave

    Wave wOut = $newName                    // Create wave reference for output wave
    wOut += 1                               // Use wave reference in assignment statement
End
```

This function might be called from the command line or from another function like this:

```
Make/O/N=5 wave0 = p
Test(wave0)          // Pass wave reference to Test function
```

A Wave statement has both a compile-time and a runtime effect.

At compile time, it tells Igor what type of object the declared name references. In the example above, it tells Igor that wOut references a wave as opposed to a numeric variable, a string variable, a window or some other type of object. The Igor compiler allows wOut to be used in a waveform assignment statement (wOut += 1) because it knows that wOut references a wave.

The compiler also needs to know if the wave is real, complex or text. Use Wave/C to create a complex wave reference and Wave/T to create a text wave reference. Wave by itself creates a real wave reference.

At runtime the Wave statement stores a reference to a specific wave in the wave reference variable (wOut in this example). The referenced wave must already exist when the wave statement executes. Otherwise Igor will store a NULL reference in the wave reference variable and you will get an error when you attempt to use it. We put the "Wave wOut = \$newName" statement *after* the Duplicate operation to insure that the wave exists when the Wave statement is executed. Putting the Wave statement before the command that creates the wave is a common error.

### Automatic Creation of WAVE References

The Igor compiler sometimes automatically creates WAVE references. For example:

```
Function Example1()
    Make wave1
    wave1= x^2
End
```

In this example we did not use a WAVE reference and yet we were still able to compile an assignment statement referring to a wave which will not exist until runtime. This is a feature of the **Make** operation (see page V-366) which automatically creates local references for simple object names. The **Duplicate** operation (see page V-133) and many other operations that create output waves also automatically create local wave references for simple object names.

Simple object names are names which are known at compile time for objects which will be created in the current data folder at runtime. Make and Duplicate do not create references if you use \$<name>, a partial

data folder path or a full data folder path to specify the object. However, as of Igor Pro 6.1, you can append `/WAVE=<name>` after a `$` or path specification to explicitly create a WAVE reference.

In the case of `Make` and `Duplicate` with simple object names, the type of the automatically created wave reference (real, complex or text) is determined by flags. `Make/C` and `Duplicate/C` create complex wave references. `Make/T` and `Duplicate/T` create text wave references. `Make` and `Duplicate` without type flags create real wave references. See **WAVE Reference Types** on page IV-58 and **WAVE Reference Type Flags** on page IV-58 for a complete list of type flags and further details.

Most built-in operations that create output waves (often called "destination" waves) also automatically create wave references. For example, if you write:

```
DWT srcWave, destWave
```

it is as if you wrote:

```
DWT srcWave, destWave
WAVE destWave
```

After the discrete wavelet transform executes, you can reference `destWave` without an explicit wave reference.

### Standalone WAVE Reference Statements

In cases where Igor does not automatically create a wave reference, because the output wave is not specified using a simple object name, you need to explicitly create a wave reference if you want to access the wave in an assignment statement.

You can create an explicit standalone wave reference using a statement following the command that created the output wave. In this example, the name of the output wave is specified as a parameter and therefore we can not use a simple object name when calling `Make`:

```
Function Example2(nameForOutputWave)
    String nameForOutputWave // String contains the name of the wave to make

    Make $nameForOutputWave // Make a wave
    Wave w = $nameForOutputWave // Make a wave reference
    w = x^2
End
```

If you make a text wave or a complex wave, you need to tell the Igor compiler about that by using `Wave/T` or `Wave/C`. The compiler needs to know the type of the wave in order to properly compile the assignment statement.

### Inline WAVE Reference Statements

In Igor Pro 6.1 or later, you can create a wave reference variable using `/WAVE=<name>` in the command that creates the output wave. For example:

```
Function Example3(nameForOutputWave)
    String nameForOutputWave

    Make $nameForOutputWave/Wave=w // Make a wave and a wave reference
    w = x^2
End
```

Here `/Wave=w` is an inline wave reference statement. It does the same thing as the standalone wave reference in the preceding section.

Here are some more examples of inline wave declarations:

```
Function Example4()
    String name = "wave1"
    Duplicate/O wave0, $name/WAVE=wave1
```

## Chapter IV-3 — User-Defined Functions

---

```
Differentiate wave0 /D=$name/WAVE=wave1
End
```

When using an inline wave reference statement, you do not need to, and in fact can not, specify the type of the wave using Wave/T or Wave/C. Just use Wave by itself regardless of the type of the output wave. The Igor compiler automatically creates the right kind of wave reference. For example:

```
Function Example5()
    Make real1, $"real2"/WAVE=r2      // real1, real2 and r2 are real
    Make/C cplx1, $"cplx2"/WAVE=c2    // cplx1, cplx2 and c2 are complex
    Make/T text1, $"text2"/WAVE=t2    // text1, text2 and t2 are text
End
```

Inline wave reference statements are accepted by those operations which automatically create a wave reference for a simple object name.

Inline wave references are not allowed after a simple object name.

Inline wave references are allowed on the command line but do nothing.

### WAVE Reference Types

When WAVE references are created at compile time, they are created with a specific numeric type or are defined as text. The compiler then uses this type when creating expressions based on the WAVE reference or when trying to match two instances of the same name. For example:

```
Make rWave
Make/C cWave
Make/T tWave
```

creates single precision real, single precision complex, and text WAVE reference variables. These types then define what kind of right-hand side expression to compile:

```
rWave= expression      // will be real
cWave= expression      // will be complex
tWave= expression      // will be text
```

At the present time, these are the only three types of expressions that the compiler creates and numeric expressions are always evaluated in double precision. However, in the future, the compiler might be extended to create integer or single precision code. As a result of these considerations, the compiler is sometimes quite picky about the exact congruence between two declarations of WAVE reference variables of the same name (which create just a single instance). For example:

```
WAVE aWave
if( !WaveExists(aWave) )
    Make/D aWave
endif
```

will give a compile error complaining about inconsistent type for a WAVE reference. This is because the default type for the WAVE reference is single precision real. In this case, you will need to use the /D flag like so:

```
WAVE/D aWave
```

### WAVE Reference Type Flags

The WAVE reference (see page V-722) along with certain operations such as Duplicate can accept the following flags identifying the type of WAVE reference:

```
/B      8-bit signed integer destination waves, unsigned with /U.
/C      complex destination waves.
```

/D	double precision destination waves.
/I	32-bit signed integer destination waves, unsigned with /U.
/S	single precision destination waves.
/T	text destination waves.
/U	unsigned destination waves.
/W	16-bit signed integer destination waves, unsigned with /U.
/DF	Wave holds data folder references. For advanced programmers only running Igor Pro 6.1 or later.
/WAVE	Wave holds wave references. For advanced programmers only running Igor Pro 6.1 or later.

These are the same flags used by the **Make** operation (see page V-366) except in this case they do not affect the actual wave and are used only specify what kind of wave is expected at runtime. This information is used if, later in the function, you create a wave assignment statement using a duplicated wave as the destination:

```
Function DupIt(wv)
    WAVE/C wv                // complex wave

    Duplicate/O/C wv,dupWv   // dupWv is complex
    dupWv[0]=cplx(5.0,1.0)   // no error, because dupWv known complex
    . . .
End
```

If Duplicate did not have the /C flag, you would get a “function not available for this number type” message when compiling the assignment of dupWv to the result of the cplx function.

### Problems with Automatic Creation of WAVE References

Operations that change a wave's type or which can create output waves of more than one type, such as **FFT**, **IFFT** and **WignerTransform** present special issues.

In some cases, the wave reference automatically created by an operation might be of the wrong type. For example, the FFT operation automatically creates a complex wave reference for the destination wave, so if you write:

```
FFT/DEST=destWave srcWave
```

it is as if you wrote:

```
FFT/DEST=destWave srcWave
WAVE/C destWave
```

However, if a real wave reference for the same name already exists, FFT/DEST will not create a new wave reference. For example:

```
Wave destWave                // Real wave reference
. . .
FFT /DEST=destWave srcWave   // FFT does not create wave reference
                             // because it already exists.
```

In this case, you would need to create a complex wave reference using a different name, like this:

```
Wave/C cDestWave = destWave
```

The output of the FFT operation can sometimes be real, not complex. For example:

```
FFT/DEST=destWave/OUT=2 srcWave
```

## Chapter IV-3 — User-Defined Functions

---

The `/OUT=2` flag creates a real destination wave. Thus the complex wave reference automatically created by the FFT operation is wrong and can not be used to subsequently access the destination wave. In this case, you must explicitly create a real wave reference, like this:

```
FFT/DEST=destWave/OUT=2 srcWave
WAVE realDestWave = destWave
```

Note that you can not write:

```
FFT/DEST=destWave/OUT=2 srcWave
WAVE destWave
```

because the FFT operation has already created a complex wave reference named `destWave`, so the compiler will generate an error. You must use a different name for the real wave reference.

The **IFFT** has a similar problem but in reverse. IFFT automatically creates a real wave reference for the destination wave. In some cases, the actual destination wave will be complex and you will need to create an explicit wave reference in order to access it.

### WAVE Reference Is Not Needed to Pass a Wave Parameter

You don't need to use a WAVE reference when passing a literal wave name as a parameter to an operation or function. For example:

```
Function Test()
    Display jack
    Display root:jack
    Variable tmp = mean(jack,0,100)
End
```

In these examples, Igor knows that the parameters of the Display operation and the first parameter of the mean function are waves and therefore does not need the additional information provided by the WAVE reference.

### Wave Reference Function Results

In Igor Pro 6.1 or later, advanced programmers can create functions that return wave references using `Function/WAVE`:

```
Function/WAVE Test(wIn) // /WAVE flag says function returns wave reference
    Wave wIn // Reference to the input wave received as parameter

    String newName = NameOfWave(wIn) + "_out" // Compute output wave name

    Duplicate/O wIn, $newName // Create output wave

    Wave wOut = $newName // Create wave reference for output wave
    wOut += 1 // Use wave reference in assignment statement

    return wOut // Return wave reference
End
```

This function might be called from another function like this:

```
Make/O/N=5 wave0 = p
Wave wOut = Test(wave0)
Display wave0, wOut
```

### Wave Reference Waves

In Igor Pro 6.1 or later, you can create waves that contain wave references using the `Make /WAVE` flag. You can use a wave reference wave as a list of waves for further processing and in multithreaded wave assignment using the **MultiThread** keyword.

Wave reference waves are recommended for advanced programmers only.

**Note:** Wave reference waves are saved only in packed experiment files. They are not saved in unpacked experiments and are not saved by the SaveData operation or the Data Browser's Save Copy button. In general, they are intended for temporary computation purposes only.

Here is an example:

```
Make/O wave0, wave1, wave2           // Make some waves
Make/O/WAVE wr                       // Make a wave reference wave
wr[0]=wave0; wr[1]=wave1; wr[2]=wave2 // Assign values
```

The wave reference wave wr could now be used, for example, to pass a list of waves to a function that performs display or analysis operations.

Make/WAVE without any assignment creates a wave containing null wave references. Similarly, inserting points or redimensioning to a larger size initializes the new points to null. Deleting points or redimensioning to a smaller size deletes any free waves if the deleted points contained the only reference to them.

To determine if a given wave is a type that stores wave references, use the **WaveType** function with the optional selector = 1.

In the next example, a subroutine supplies a list of references to waves to be graphed by the main routine. A wave reference wave is used to store the list of wave references.

```
Function MainRoutine()
  Make/O/WAVE/N=5 wr           // Will contain references to other waves
  wr= Subroutine(p)           // Fill w with references

  WAVE w= wr[0]               // Get reference to first wave
  Display w                   // and display in a graph

  Variable i
  for(i=1;i<5;i+=1)
    WAVE w= wr[i]             // Get reference to next wave
    AppendToGraph w           // and append to graph
  endfor
End

Function/WAVE Subroutine(i)
  Variable i

  String name = "wave"+num2str(i)

  // Create a wave with a computed name and also a wave reference to it
  Make/O $name/WAVE=w = sin(x/(8+i))

  return w                    // Return the wave reference to the calling routine
End
```

For an example using a wave reference wave for multiprocessing, see **Wave Reference MultiThread Example** on page IV-286.

## Data Folder References

The data folder reference is a lightweight object that refers to a data folder, analogous to the wave reference which refers to a wave. You can think of a data folder reference as an identification number that Igor uses to identify a particular data folder.

Data folder references require Igor Pro 6.1 or later.

Data folder reference variables (DFREFs) hold data folder references. They can be created as local variables, passed as parameters and returned as function results.

## Chapter IV-3 — User-Defined Functions

---

The most common use for a data folder reference is to save and restore the current data folder during the execution of a function:

```
Function Test()  
    DFREF savedFR = GetDataFolderDFR() // Get reference to current data folder  
  
    NewDataFolder/O/S MyNewDataFolder // Create a new data folder  
    . . . // Do some work in it  
  
    SetDataFolder savedFR // Restore current data folder  
End
```

Data folder references can be used in commands where Igor accepts a data folder path. For example, this function shows three equivalent methods of accessing waves in a specific data folder:

```
Function Test()  
    // Method 1: Using paths  
    Display root:Packages:'My Package':yWave vs root:Packages:'My Package':xWave  
  
    // Method 2: Using the current data folder  
    String dfSave = GetDataFolder(1) // Save the current data folder  
    SetDataFolder root:Packages:'My Package'  
    Display yWave vs xWave  
    SetDataFolder dfSave // Restore current data folder  
  
    // Method 3: Using data folder references  
    DFREF dfr = root:Packages:'My Package'  
    Display dfr:yWave vs dfr:xWave  
End
```

Using data folder references instead of data folder paths can streamline programs that make heavy use of data folders.

### Using Data Folder References

In an advanced application, the programmer often defines a set of named data objects (waves, numeric variables and string variables) that the application acts on. These objects exist in a data folder. If there is just one instance of the set, it is possible to hard-code data folder paths to the objects. Often, however, there will be a multiplicity of such sets, for example, one set per graph or one set per channel in a data acquisition application. In such applications, procedures must be written to act on the set of data objects in a data folder specified at runtime.

One way to specify a data folder at runtime is to create a path to the data folder in a string variable. While this works, you wind up with code that does a lot of concatenation of data folder paths and data object names. Using data folder references, such code can be streamlined.

You create a data folder reference variable with a DFREF statement. For example, assume your application defines a set of data with a wave named wave0, a numeric variable named num0 and a string named str0 and that we have one data folder containing such a set for each graph. You can access your objects like this:

```
Function DoSomething(graphName)  
    String graphName  
    DFREF dfr = root:Packages:MyApplication:$graphName  
    WAVE w0 = dfr:wave0  
    NVAR n0 = dfr:num0  
    SVAR s0 = dfr:str0  
    . . .  
End
```

Igor accepts a data folder reference in any command in which a data folder path would be accepted. For example:

```

Function Test()
  Display root:MyDataFolder:wave0 // OK

  DFREF dfr = root:MyDataFolder
  Display dfr:wave0 // OK

  String path = "root:MyDataFolder:wave0"
  Display $path // OK. $ converts string to path.

  path = "root:MyDataFolder"
  DFREF dfr = $path // OK. $ converts string to path.
  Display dfr:wave0 // OK

  String currentDFPath
  currentDFPath = GetDataFolder(1) // OK
  DFREF dfr = GetDataFolder(1) // ERROR: GetDataFolder returns a string
  // not a path.
End

```

### The /SDFR Flag

In Igor Pro 6.20 or later you can also use the /SDFR (source data folder reference) flag in a WAVE, NVAR or SVAR statement. The utility of /SDFR is illustrated by this example which shows three different ways to reference multiple waves in the same data folder:

```

Function Test()
  // Assume a data folder exists at root:Run1

  // Use explicit paths
  Wave wave0=root:Run1:wave0, wave1=root:Run1:wave1, wave2=root:Run1:wave2

  // Use a data folder reference
  DFREF dfr = root:Run1
  Wave wave0=dfr:wave0, wave1=dfr:wave1, wave2=dfr:wave2

  // Use the /SDFR flag
  DFREF dfr = root:Run1
  Wave/SDFR=dfr wave0, wave1, wave2
End

```

### The DFREF Type

In Functions, you can define data folder reference variables using the DFREF declaration:

```
DFREF localname [= <DataFolderRef or path>] [<more defs>]
```

You can then use the data folder reference in those places where you can use a data folder path. For example:

```
DFREF dfr= root:df1
Display dfr:wave1 // Equivalent to Display root:df1:wave1
```

The syntax is limited to a single name after the data folder reference, so this is not legal:

```
Display dfr:subfolder:wave1 // Illegal
```

You can use DFREF to define input parameters for user-defined functions. For example:

```

Function Test(df)
  DFREF df
  Display df:wave1
End

```

## Chapter IV-3 — User-Defined Functions

---

You can also use DFREF to define fields in structures. However, you can not directly use a DFREF structure field in those places where Igor is expecting a path and object name. So, instead of:

```
Display s.dfr:wave1           // Illegal
```

you would need to write:

```
DFREF dftmp= s.dfr
Display dftmp:wave1         // OK
```

You can use a DFREF structure field where just a path is expected. For example:

```
SetDataFolder s.dfr         // OK
```

### Built-in DFREF Functions

Some built-in functions take string data folder paths as parameters or return them as results. Those functions can not take or return data folder references. Here are equivalent DFREF versions that take or return data folder references:

```
GetDataFolderDFR()
GetIndexedObjNameDFR(dfr, type, index)
GetWavesDataFolderDFR(wave)
CountObjectsDFR(dfr, type)
```

These additional data folder reference functions are available:

```
DataFolderRefStatus(dfr)
NewFreeDataFolder()
DataFolderRefsEqual(dfr1, dfr2)
```

Just as operations that take a data folder path accept a data folder reference, these DFREF functions can also accept a data folder path:

```
Function Test()
  DFREF dfr = root:MyDataFolder
  Print CountObjectsDFR(dfr,1)           // OK
  Print CountObjectsDFR(root:MyDataFolder,1) // OK
End
```

### Checking Data Folder Reference Validity

The **DataFolderRefStatus** function returns zero if the data folder reference is invalid. You should use it to test any DFREF variables that might not be valid, for example, when you assign a value to a data folder reference and you are not sure that the referenced data folder exists:

```
Function Test()
  DFREF dfr = root:MyDataFolder // MyDataFolder may or may not exist
  if (DataFolderRefStatus(dfr) != 0)
    . . .
  endif
End
```

For historical reasons, an invalid DFREF variable will often act like root.

### Data Folder Reference Function Results

A user-defined function can return a data folder reference. This might be used for a subroutine that returns a set of new objects to the calling routine. The set can be returned in a new data folder and the subroutine can return a reference it.

For example:

```
Function/DF Subroutine(newDFName)
  String newDFName
  NewDataFolder/O $newDFName
  DFREF dfr = $newDFName
  Make/O dfr:wave0, dfr:wave1
  return dfr
End

Function/DF MainRoutine()
  DFREF dfr = Subroutine("MyDataFolder")
  Display dfr:wave0, dfr:wave1
End
```

## Data Folder Reference Waves

In Igor Pro 6.1 or later, you can create waves that contain data folder references using the **Make /DF** flag. You can use a data folder reference wave as a list of data folders for further processing and in multithreaded wave assignment using the **MultiThread** keyword.

Data folder reference waves are recommended for advanced programmers only.

**Note:** Data folder reference waves are saved only in packed experiment files. They are not saved in unpacked experiments and are not saved by the **SaveData** operation or the Data Browser's **Save Copy** button. In general, they are intended for temporary computation purposes only.

**Make/DF** without any assignment creates a wave containing null data folder references. Similarly, inserting points or redimensioning to a larger size initializes the new points to null. Deleting points or redimensioning to a smaller size deletes any free data folders if the wave contained the only reference to them.

To determine if a given wave is a type that stores data folder references, use the **WaveType** function with the optional selector = 1.

Although programmers may find multiple uses of data folder reference waves, the main impetus for their development was to make it easier to improve calculation performance by multithreading on multi-core machines.

For an example using a data folder reference wave for multiprocessing, see **Data Folder Reference Multi-Thread Example** on page IV-285.

## Accessing Waves in Functions

To access a wave, we need to create, one way or another, a wave reference. The section **Accessing Global Variables and Waves** on page IV-50 explained how to access a wave using a **WAVE** reference. This section introduces several additional techniques. Other useful details may be found in **Returning Created Waves from User Functions** on page III-142.

We can create the wave reference by:

- Declaring a wave parameter
- Using `$<string expression>`
- Using a literal wave name
- Using a wave reference function

Each of these techniques is illustrated in the following sections.

Each example shows a function and commands that call the function. The function itself illustrates how to deal with the wave within the function. The commands show how to pass enough information to the function so that it can access the wave. Other examples can be found in **Writing Functions that Process Waves** on page III-143.

## Chapter IV-3 — User-Defined Functions

---

### Wave Reference Passed as Parameter

This is the simplest method. The function might be called from the command line or from another function.

```
Function Test(w)
    WAVE w // Wave reference passed as a parameter

    w += 1 // Use in assignment statement
    Print mean(w, -INF, INF) // Pass as function parameter
    WaveStats w // Pass as operation parameter
    AnotherFunction(w) // Pass as user function parameter
End

Make/O/N=5 wave0 = p
Test(wave0)
Test($"wave0")
String wName = "wave0"; Test($wName)
```

In the first call to Test, the wave reference is a literal wave name. In the second call, we create the wave reference using `$<literal string>`. In the third call, we create the wave reference using `$<string variable>`. `$<literal string>` and `$<string variable>` are specific cases of the general case `$<string expression>`.

If the function expected to receive a reference to a text wave, we would declare the parameter using:

```
WAVE/T w
```

If the function expected to be receive a reference to a complex wave, we would declare the parameter using:

```
WAVE/C w
```

If you need to return a large number of values to the calling routine, it is sometimes convenient to use a parameter wave as an output mechanism. The following example illustrates this technique:

```
Function MyWaveStats(inputWave, outputWave)
    WAVE inputWave
    WAVE outputWave

    WaveStats/Q inputWave

    outputWave[0] = V_npnts
    outputWave[1] = V_avg
    outputWave[2] = V_sdev
End

Function Test()
    Make/O testwave= gnoise(1)

    Make/O/N=20 tempResultWave
    MyWaveStats(testwave, tempResultWave)
    Variable npnts = tempResultWave[0]
    Variable avg = tempResultWave[1]
    Variable sdev = tempResultWave[2]
    KillWaves tempResultWave

    Printf "Points: %g; Avg: %g; SDev: %g\r", npnts, avg, sdev
End
```

### Wave Accessed Via String Passed as Parameter

This technique is of most use when the wave might not exist when the function is called. It is appropriate for functions that create waves.

```
Function Test(wName)
    String wName // String containing a name for wave

    Make/O/N=5 $wName
    WAVE w = $wName // Create a wave reference
    Print NameOfWave(w)
```

End

```
Test ("wave0")
```

This example will create wave0 if it does not yet exist or overwrite it if it does exist. If we knew that the wave had to already exist, we could and should use the wave parameter technique shown in the preceding section. In this case, since the wave may not yet exist, we can not use a wave parameter.

Notice that we create a wave reference immediately after making the wave. Once we do this, we can use the wave reference in all of the ways shown in the preceding section. We can not create the wave reference before making the wave because a wave reference must refer to an existing wave.

The following command demonstrates that \$wName and the wave reference w can refer to a wave that is not in the current data folder.

```
NewDataFolder root:Folder1; Test("root:Folder1:wave0")
```

### Wave Accessed Via String Calculated in Function

This technique is used when creating multiple waves in a function or when algorithmically selecting a wave or a set of waves to be processed.

```
Function Test(baseName, startIndex, endIndex)
    String baseName
    Variable startIndex, endIndex

    Variable index = startIndex
    do
        WAVE w = $(baseName + num2istr(index))
        WaveStats/Q w
        Printf "Wave: %s; average: %g\r", NameOfWave(w), V_avg
        index += 1
    while (index <= endIndex)
End
```

```
Make/O/N=5 wave0=gnoise(1), wave1=gnoise(1), wave2=gnoise(1)
Test("wave", 0, 2)
```

We need to use this method because we want the function to operate on any number of waves. If the function were to operate on a small, fixed number of waves, we could use the wave parameter method.

As in the preceding section, we create a wave reference using \$<string expression>.

### Wave Accessed Via Literal Wave Name

In data acquisition or analysis projects, you often need to write procedures that deal with runs of identically-structured data. Each run is stored in its own data folder and contains waves with the same names. In this kind of situation, you can write a set of functions that use literal wave names specific for your data structure.

```
Function CreateRatio()
    WAVE dataA, dataB
    Duplicate dataA, ratioAB
    WAVE ratioAB
    ratioAB = dataA / dataB
End
```

```
Make/O/N=5 dataA = 1 + p, dataB = 2 + p
CreateRatio()
```

The CreateRatio function assumes the structure and naming of the data. The function is hard-wired to this naming scheme and assumes that the current data folder contains the appropriate data.

## Chapter IV-3 — User-Defined Functions

---

### Wave Accessed Via Wave Reference Function

A wave reference function is a built-in Igor function that returns a reference to a wave. Wave reference functions are typically used on the right-hand side of a WAVE statement. For example:

```
WAVE w = WaveRefIndexed("", i, 4)
```

A common use for a wave reference function is to get access to waves displayed in a graph, using the TraceNameToWaveRef function. Here is an example.

```
Function PrintAverageOfDisplayedWaves()  
    String list, traceName  
  
    list = TraceNameList("", "", 1) // List of traces in top graph  
    Variable index = 0  
    do  
        traceName = StringFromList(index, list) // Next trace name  
        if (strlen(traceName) == 0)  
            break // No more traces  
        endif  
        WAVE w = TraceNameToWaveRef("", traceName) // Get wave ref  
        WaveStats/Q w  
        Printf "Wave: %s; average: %g\r", NameOfWave(w), V_avg  
        index += 1  
    while (1) // loop till break above  
End
```

```
Make/O/N=5 wave0=gnoise(1), wave1=gnoise(1), wave2=gnoise(1)  
Display wave0, wave1, wave2  
PrintAverageOfDisplayedWaves()
```

There are other wave reference functions (see **Wave Reference Functions** on page IV-173), but WaveRefIndexed and TraceNameToWaveRef are the most used.

### Destination Wave Parameters

Many operations create waves. Examples are Make, Duplicate and Differentiate. Such operations take "destination wave" parameters. A destination wave parameter can be:

A simple name	Differentiate fred /D=jack
A path	Differentiate fred /D=root:FolderA:jack
\$ followed by a string expression	String str = "root:FolderA:jack" Differentiate fred /D=\$str
A simple wave reference	Wave/Z jack Differentiate fred /D=jack
Other wave references	Wave/Z w = jack Differentiate fred /D=w  Wave/Z w = root:FolderA:jack Differentiate fred /D=w  Wave/Z jack = root:FolderA:jack Differentiate fred /D=jack  STRUCT MyStruct s // Contains wave ref field w Differentiate fred /D=s.w

The wave reference works only in a user-defined function. The other techniques work in functions, in macros and from the command line.

The first and fourth cases are actually the same because, if you use a simple name, the Igor compiler automatically creates a wave reference as if you had used a simple wave reference as explained under **Automatic Creation of WAVE References** on page IV-56.

Using the first four techniques, the destination wave may or may not already exist. It will be created if it does not exist and overwritten if it does exist.

In the last technique, the name of the wave reference is different from the name of the destination wave or the wave reference points to a data folder other than the current data folder or both. This technique works properly only if the referenced wave already exists. Otherwise the destination wave is a wave with the name of the wave reference itself (w in the first two examples, jack in the last) in the current data folder. This situation is further explained below under **Destination Wave Reference Issues** on page IV-70.

### Wave Reference as Destination Wave

Here are the rules for wave references when used as destination waves:

1. If a simple name (not a wave reference) is passed as the destination wave parameter, the destination wave is a wave of that name in the current data folder whether it exists or not.
2. If a path or \$ followed by a string containing a path is passed as the destination wave parameter, the destination wave is specified by the path whether the specified wave exists or not.
3. If a wave reference is passed as the destination wave parameter, the destination wave is the referenced wave if it exists. See **Destination Wave Reference Issues** on page IV-70 for what happens if it does not exist.

### Exceptions To Destination Wave Rules

The Make operation is an exception in regard to wave references. The following statements make a wave named w in the current data folder whether root:FolderA:jack exists or not:

```
Wave/Z w = root:FolderA:jack
Make/O w
```

Prior to Igor Pro 6.20, many operations behaved like Make in this regard. In Igor Pro 6.20 and later, most operations behave like Differentiate.

### Updating of Destination Wave References

When a simple name is provided in a user-defined function, Igor automatically creates a wave reference variable of that name at compile time if one does not already exist. This is an "implicit" wave reference.

When a wave reference variable exists, whether implicit or explicit, during the execution of the command, the operation stores a reference to the destination wave in that wave reference variable. You can use the wave reference to access the destination wave in subsequent commands.

Similarly, when the destination is specified using a wave reference field in a structure, the operation updates the field to refer to the destination wave.

### Inline Wave References With Destination Waves

When the destination wave is specified using a path or a string containing a path, you can use an inline wave reference to create a reference to the destination wave. For example:

```
String str = "root:FolderA:jack"
Differentiate $str /D=$dest/WAVE=wDest
Display wDest
```

Here the Igor compiler creates a wave reference named wDest. The Differentiate operation stores a reference to the destination wave (root:FolderA:jack in this case) in wDest which can then be used to access the destination wave in subsequent commands.

## Chapter IV-3 — User-Defined Functions

---

Inline wave references do not determine which wave is the destination wave but merely provide a wave reference pointing to the destination wave when the command finishes.

### Destination Wave Reference Issues

You will get unexpected behavior when a wave reference variable refers to a wave with a different name or in a different data folder and the referenced wave does not exist. For example, if the referenced wave does not exist:

```
Wave/Z w = jack
Differentiate fred /D=w      // Creates a wave named w in current data folder

Wave/Z w = root:FolderA:jack
Differentiate fred /D=w      // Creates a wave named w in current data folder

Wave/Z jack = root:FolderA:jack
Differentiate fred /D=jack // Creates a wave named jack in current data folder

STRUCT MyStruct s          // Contains wave ref field w
Differentiate fred /D=s.w  // Creates a wave named w in current data folder
```

In a situation like this, you should add a test using `WaveExists` to verify that the destination wave is valid and throw an error if not or otherwise handle the situation. For example:

```
Wave/Z w = root:FolderA:jack
if (!WaveExists(w))
    Abort "Destination wave does not exist"
endif
Differentiate fred /D=w
```

As noted above, when you use a simple name as a destination wave, the Igor compiler automatically creates a wave reference. If the automatically-created wave reference conflicts with a pre-existing wave reference, the compiler will generate an error. For example, this function generates an "inconsistent type for wave reference error":

```
Function InconsistentTypeError()
    Wave/C w          // Explicit complex wave reference
    Differentiate fred /D=w // Implicit real wave reference
End
```

Another consideration involves loops. Suppose in a loop you have code like this:

```
SetDataFolder <something depending on loop index>
Duplicate/O srcWave, jack
```

You may think you are creating a wave named `jack` in each data folder but, because the contents of the automatically-created wave reference variable `jack` is non-null after the first iteration, you will simply be overwriting the same wave over and over. To fix this, use

```
Duplicate/O srcWave,$"jack"/WAVE=jack
```

This will create a wave named `jack` in the current data folder and store a reference to it in a wave reference variable also named `jack`.

### Changes in Destination Wave Behavior

Igor's handling of destination wave references was improved for Igor Pro 6.20. Previously some operations treated wave references as simple names, did not set the wave reference to refer to the destination wave on output, and exhibited other non-standard behavior.

## Trace Name Parameters

The `ModifyGraph` operation takes trace name parameters. A trace name is not the same as a wave. An example may clarify this subtle point:

```
Function Test()
    Wave w = root:FolderA:wave0
    Display w
    ModifyGraph rgb(w) = (65535,0,0)          // WRONG
End
```

This is wrong because `ModifyGraph` is looking for the name of a trace in a graph and `w` is not the name of a trace in a graph. The name of the trace in this case is `wave0`. The function should be written like this:

```
Function Test()
    Wave w = root:FolderA:wave0
    Display w
    ModifyGraph rgb(wave0) = (65535,0,0)     // CORRECT
End
```

In the next example, the wave is passed to the function as a parameter so the name of the trace is not so obvious:

```
Function Test(w)
    Wave w
    Display w
    ModifyGraph rgb(w) = (65535,0,0)        // WRONG
End
```

This is wrong for the same reason as the first example: `w` is not the name of the trace. The function should be written like this:

```
Function Test(w)
    Wave w
    Display w

    String name = NameOfWave(w)
    ModifyGraph rgb($name) = (65535,0,0)    // CORRECT
End
```

## User-defined Trace Names

As of Igor Pro 6.20, you can provide user-defined names for traces using `/TN=<name>` with `Display` and `AppendToGraph`. For example:

```
Make/O jack=sin(x/8)
NewDataFolder/O foo; Make/O :foo:jack=sin(x/9)
NewDataFolder/O bar; Make/O :bar:jack=sin(x/10)
Display jack/TN='jack in root', :foo:jack/TN='jack in foo'
AppendToGraph :bar:jack/TN='jack in bar'
ModifyGraph mode('jack in bar')=7,hbFill('jack in bar')=6
ModifyGraph rgb('jack in bar')=(0,0,65535)
```

**Warning:** Using this feature will make your experiment hard to load into versions prior to 6.20. Also, any code that assumes the trace name contains the wave name will fail.

## Free Waves

Free waves are waves that are not part of any data folder hierarchy. Their principal use is in multithreaded wave assignment using the `MultiThread` keyword in a function. They can also be used for temporary storage within functions.

Free waves are recommended for advanced programmers only.

## Chapter IV-3 — User-Defined Functions

---

Free waves require Igor Pro 6.1 or later.

**Note:** Free waves are saved only in packed experiment files. They are not saved in unpacked experiments and are not saved by the SaveData operation or the Data Browser's Save Copy button. In general, they are intended for temporary computation purposes only.

You create free waves using the **NewFreeWave** function and the **Make/FREE** and **Duplicate/FREE** operations.

Here is an example:

```
Function ReverseWave(w)
    Wave w

    Variable lastPoint = numpnts(w) - 1
    if (lastPoint > 0)
        // This creates a free wave named wTemp.
        // It also creates an automatic wave reference named wTemp
        // which refers to the free wave.
        Duplicate /FREE w, wTemp

        w = wTemp[lastPoint-p]
    endif
End
```

In this example, wTemp is a free wave. As such, it is not contained in any data folder and therefore can not conflict with any other wave.

As explained below under **Free Wave Lifetime** on page IV-73, a free wave is automatically deleted when there are no more references to it. In this example that happens when the function ends.

You can access a free wave only using the wave reference returned by NewFreeWave, Make/FREE or Duplicate/FREE.

Free waves can not be used in situations where global persistence is required such as in graphs, tables and controls. In other words, you should use free waves for computation purposes only.

For a discussion of multithreaded assignment statements, see **Automatic Parallel Processing with Multi-Thread** on page IV-283. For an example using free waves, see **Wave Reference MultiThread Example** on page IV-286.

### Free Wave Created When Free Data Folder Is Deleted

In addition to the explicit creation of free waves, a wave that is created in a free data folder becomes free if the host data folder is deleted but a reference to the wave still exists. For example:

```
Function/WAVE Test()
    DFREF dfSav= GetDataFolderDFR()

    SetDataFolder NewFreeDataFolder()
    Make jack={1,2,3} // jack is not a free wave at this point.
    // This also creates an automatic wave reference named jack.

    SetDataFolder dfSav
    // The free data folder is now gone but jack persists
    // because of the wave reference to it and is now a free wave.

    return jack
End
```

## Free Wave Created For User Function Input Parameter

If a user function takes a wave reference as in input parameter, you can create and pass a short free wave using a list of values as illustrated here:

```
Function Test(w)
    WAVE/D w
    Print w
End
```

You can invoke this function like this:

```
Test({1, 2, 3})
```

Igor automatically creates a free wave and passes it to Test. The free wave is automatically deleted when Test returns.

The data type of the free wave is determined by the type of the function's wave parameter. In this case the free wave is created as double-precision floating point because the wave parameter is defined using the /D flag. If /D were omitted, the free wave would be single-precision floating point. Wave/C/D would give a double-precision complex free wave. Wave/T would give a text free wave.

This list of values syntax is allowed only for user-defined functions because only they have code to test for and delete free waves upon exit.

## Free Wave Lifetime

A free wave is automatically deleted when the last reference to it disappears.

Wave references can be stored in:

1. Wave reference variables in user-defined functions
2. Wave reference fields in structures
3. Elements of a wave reference wave (created by Make/WAVE)

The first case is the most common.

A wave reference disappears when:

1. The wave reference variable containing it is explicitly cleared using WaveClear.
2. The wave reference variable containing it is reassigned to refer to another wave.
3. The wave reference variable containing it goes out-of-scope and ceases to exist when the function in which it was created returns.
4. The wave reference wave element containing it is deleted or the wave reference wave is killed.

When there are no more references to a free wave, Igor automatically deletes it. This example illustrates the first three of these scenarios:

```
Function TestFreeWaveDeletion1()
    Wave w = NewFreeWave(2,3) // Create a free wave with 3 points
    WaveClear w              // w no longer refers to the free wave
    // There are no more references to the free wave so it is deleted

    Wave w = NewFreeWave(2,3) // Create a free wave with 3 points
    Wave w = root:wave0       // w no longer refers to the free wave
    // There are no more references to the free wave so it is deleted

    Wave w = NewFreeWave(2,3) // Create a free wave with 3 points
End // Wave reference w ceases to exist so the free wave is deleted
```

In the preceding example we used NewFreeWave which creates a free wave named 'f' that is not part of any data folder. Next we will use Make/FREE instead of NewFreeWave. Using Make allows us to give the free

## Chapter IV-3 — User-Defined Functions

---

wave a name of our choice but it is still free and not part of any data folder. When reading this example, keep in mind that "Make jack" creates an automatic wave reference variable named jack:

```
Function TestFreeWaveDeletion2()  
    Make /D /N=3 /FREE jack    // Create a free DP wave with 3 points  
    // Make created an automatic wave reference named jack  
  
    Make /D /N=5 /FREE jack    // Create a free DP wave with 5 points  
    // Make created an automatic wave reference named jack  
    // which refers to the 5-point jack.  
    // There are now no references to 3-point jack so it is automatically deleted.  
  
End    // Wave reference jack ceases to exist so free wave jack is deleted
```

In the next example, a subroutine returns a reference to a free wave to the calling routine:

```
Function/WAVE Subroutine1()  
    Make /D /N=3 /FREE jack=p  // Create a free DP wave with 3 points  
    return jack                // Return reference to calling routine  
End  
  
Function MainRoutine1()  
    WAVE w= Subroutine1()    // Wave reference w references the free wave jack  
    Print w  
End    // Wave reference w ceases to exist so free wave jack is deleted
```

In the final example, the wave jack starts as an object in a free data folder (see **Free Data Folders** on page IV-75). It is not free because it is part of a data folder hierarchy even though the data folder is free. When the free data folder is deleted, jack becomes a free wave.

When reading this example, keep in mind that the free data folder is automatically deleted when there are no more references to it. Originally, it survives because it is the current data folder and therefore is referenced by Igor internally. When it is no longer the current data folder, there are no more references to it and it is automatically deleted:

```
Function/WAVE Subroutine2()  
    DFREF dfSav= GetDataFolderDFR()  
  
    // Create free data folder and set as current data folder  
    SetDataFolder NewFreeDataFolder()  
  
    // Create wave jack and an automatic wave reference to it  
    Make jack={1,2,3}    // jack is not free - it is an object in a data folder  
  
    SetDataFolder dfSav // Change the current data folder  
    // There are now no references to the free data folder so it is deleted  
    // but wave jack remains because there is a reference to it.  
    // jack is now a free wave.  
  
    return jack          // Return reference to free wave to calling routine  
End  
  
Function MainRoutine2()  
    WAVE w= Subroutine2() // Wave reference w references free wave jack  
    Print w  
End    // Wave reference w ceases to exist so free wave jack is deleted
```

Not shown in this section is the case of a free wave that persists because a reference to it is stored in a wave reference wave. That situation is illustrated by the **Automatic Parallel Processing with MultiThread** on page IV-283 example.

## Converting a Free Wave to a Global Wave

You can use **MoveWave** to move a free wave into a global data folder, in which case it ceases to be free. If the wave was created by **NewFreeWave** its name will be 'f'. You can use **MoveWave** to provide it with a more descriptive name.

Here is an example illustrating **Make/FREE**:

```
Function Test()
  Make/FREE/N=(50,50) w
  SetScale x,-5,8,w
  SetScale y,-7,12,w
  w= exp(-(x^2+y^2))
  NewImage w
  if( GetRTErrors(1) != 0 )
    Print "Can't use a free wave here"
  endif
  MoveWave w,root:wasFree
  NewImage w
End
```

Note that **MoveWave** requires that the new wave name, **wasFree** in this case, be unique within the destination data folder.

To determine if a given wave is free or global, use the **WaveType** function with the optional selector = 2.

## Free Data Folders

Free data folders are data folders that are not part of any data folder hierarchy. Their principal use is in multithreaded wave assignment using the **MultiThread** keyword in a function. They can also be used for temporary storage within functions.

Free data folders are recommended for advanced programmers only.

Free data folders require Igor Pro 6.1 or later.

**Note:** Free data folders are saved only in packed experiment files. They are not saved in unpacked experiments and are not saved by the **SaveData** operation or the **Data Browser's Save Copy** button. In general, they are intended for temporary computation purposes only.

You create a free data folder using the **NewFreeDataFolder** function. You access it using the data folder reference returned by that function.

After using **SetDataFolder** with a free data folder, be sure to restore it to the previous value, like this:

```
Function Test()
  DFREF dfrSave = GetDataFolderDFR()

  SetDataFolder NewFreeDataFolder() // Create new free data folder.

  . . .

  SetDataFolder dfrSave
End
```

This is good programming practice in general but is especially important when using free data folders.

Free data folders can not be used in situations where global persistence is required such as in graphs, tables and controls. In other words, you should use objects in free data folders for short-term computation purposes only.

## Chapter IV-3 — User-Defined Functions

---

For a discussion of multithreaded assignment statements, see **Automatic Parallel Processing with Multi-Thread** on page IV-283. For an example using free data folders, see **Data Folder Reference MultiThread Example** on page IV-285.

### Free Data Folder Lifetime

A free data folder is automatically deleted when the last reference to it disappears.

Data folder references can be stored in:

1. Data folder reference variables in user-defined functions
2. Data folder reference fields in structures
3. Elements of a data folder reference wave (created with Make/DF)
4. Igor's internal current data folder reference variable

A data folder reference disappears when:

1. The data folder reference variable containing it is explicitly cleared using KillDataFolder.
2. The data folder reference variable containing it is reassigned to refer to another data folder.
3. The data folder reference variable containing it goes out-of-scope and ceases to exist when the function in which it was created returns.
4. The data folder reference wave element containing it is deleted or the data folder reference wave is killed.
5. The current data folder is changed which causes Igor's internal current data folder reference variable to refer to another data folder.

When there are no more references to a free data folder, Igor automatically deletes it.

In this example, a free data folder reference variable is cleared by KillDataFolder:

```
Function Test1()  
    DFREF dfr= NewFreeDataFolder()          // Create new free data folder.  
    // The free data folder exists because dfr references it.  
    . . .  
    KillDataFolder dfr // dfr no longer refers to the free data folder  
End
```

KillDataFolder kills the free data folder only if the given DFREF variable contains the last reference to it.

In the next example, the free data folder is automatically deleted when the DFREF that references it is changed to reference another data folder:

```
Function Test2()  
    DFREF dfr= NewFreeDataFolder()          // Create new free data folder.  
    // The free data folder exists because dfr references it.  
    . . .  
    DFREF dfr= root:  
    // The free data folder is deleted since there are no references to it.  
End
```

In the next example, a free data folder is created and a reference is stored in a local data folder reference variable. When the function ends, the DFREF ceases to exist and the free data folder is automatically deleted:

```
Function Test3()  
    DFREF dfr= NewFreeDataFolder()          // Create new free data folder.  
    // The free data folder exists because dfr references it.  
    . . .  
End // The free data folder is deleted because dfr no longer exists.
```

The fourth case, where a data folder reference is stored in a data folder reference wave, is discussed under **Data Folder Reference Waves** on page IV-65.

In the next example, the free data folder is referenced by Igor's internal current data folder reference variable because it is the current data folder. When the current data folder is changed, there are no more references to the free data folder and it is automatically deleted:

```
Function Test4()
  SetDataFolder NewFreeDataFolder() // Create new free data folder.
  // The free data folder persists because it is the current data folder
  // and therefore is referenced by Igor's internal
  // current data folder reference variable.
  . . .

  // Change Igor's internal current data folder reference
  SetDataFolder root:
  // The free data folder is since there are no references to it.
End
```

### Free Data Folder Objects Lifetime

Next we consider what happens to objects in a free data folder when the free data folder is deleted. In this event, numeric and string variables in the free data folder are unconditionally automatically deleted. A wave is automatically deleted if there are no wave references to it. If there is a wave reference to it, the wave survives and becomes a free wave. Free waves are waves that exist outside of any data folder as explained under **Free Waves** on page IV-71.

For example:

```
Function Test()
  SetDataFolder NewFreeDataFolder() // Create new free data folder.
  // The free data folder exists because it is the current data folder.

  Make jack // Make a wave and an automatic wave reference

  . . .

  SetDataFolder root:
  // The free data folder is deleted since there are no references to it.
  // Because there is a reference to the wave jack, it persists
  // and becomes a free wave.

  . . .

End // The wave reference to jack ceases to exist so jack is deleted
```

When this function ends, the reference to the wave jack ceases to exist, there are no references to jack, and it is automatically deleted.

Next we look at a slight variation. In the following example, Make does not create an automatic wave reference because of the use of \$, and we do not create an explicit wave reference:

```
Function Test()
  SetDataFolder NewFreeDataFolder() // Create new free data folder.
  // The free data folder exists because it is the current data folder.

  Make $"jack" // Make a wave but no wave reference
  // jack persists because the current data folder references it.

  . . .

  SetDataFolder root:
  // The free data folder is since there are no references to it.
  // jack is also deleted because there are no more references to it.
```

End

### Converting a Free Data Folder to a Global Data Folder

You can use **MoveDataFolder** to move a free data folder into the global hierarchy. The data folder and all of its contents then become global. The name of a free data folder created by **NewFreeDataFolder** is 'free-root'. You should rename it after moving to a global context. For example:

```
Function Test()  
  DFREF saveDF = GetDataFolderDFR()  
  DFREF dfr = NewFreeDataFolder() // Create free data folder  
  SetDataFolder dfr // Set as current data folder  
  Make jack=sin(x/8) // Create some data in it  
  SetDataFolder saveDF // Restore original current data folder  
  MoveDataFolder dfr, root: // Free DF becomes root:freeroot  
  RenameDataFolder root:freeroot,TestDF // Rename with a proper name  
  Display root:TestDF:jack  
End
```

Note that **MoveDataFolder** requires that the data folder name, **freeroot** in this case, be unique within the destination data folder.

## Structures in Functions

You can define structures in procedure files and use them in functions. Structures can be used only in user-defined functions as local variables and their behavior is defined almost entirely at compile time. Runtime or interactive definition and use of structures is not currently supported (for this purpose, use Data Folders (see Chapter II-8, **Data Folders**), the **StringByKey** function (see page V-675), or the **NumberByKey** function (see page V-458)).

Use of structures is an advanced technique. If you are just starting with Igor programming, you may want to skip this section and come back to it later.

### Defining Structures

Structures are defined in a procedure file with the following syntax:

```
Structure structureName  
  memType memName [arraySize] [, memName [arraySize]]  
  ...  
EndStructure
```

Structure member types (*memType*) can be any of the following Igor objects: Variable, String, WAVE, NVAR, SVAR, DFREF, FUNCREF, or STRUCT. The DFREF object type requires Igor Pro 6.1 or later.

Igor structures also support additional member types, as given in the next table, for compatibility with C programming structures and disk files.

Igor Member Type	C Equivalent	Byte Size
char	signed char	1
uchar	unsigned char	1
int16	short int	2
uint16	unsigned short int	2
int32	long int	4
uint32	unsigned long int	4

Igor Member Type	C Equivalent	Byte Size
float	float	4
double	double	8

The Variable and double types are identical although Variable can be also specified as complex (using the /C flag).

The optional *arraySize* must be specified using an expression involving literal numbers or locally-defined constants enclosed in brackets. The value must be between 1 and 400 for all but STRUCT where the upper limit is 100. The upper limit is a consequence of how memory is allocated on the stack frame.

Structures are two-byte aligned. This means that if an odd number of bytes has been allocated and then a nonchar field is defined, an extra byte of padding will be inserted in front of the new field. This is mainly of concern only when reading and writing structures from and to disk files.

The Structure keyword can be preceded with the **Static** keyword (see page V-594) to make the definition apply only to the current procedure window. Without the Static designation, structures defined in one procedure window may be used in any other.

## Using Structures

To use (“instantiate”) a structure in a function, you must allocate a STRUCT variable using:

```
STRUCT sName name
```

where *sName* is the name (or “tag” if you’re a C programmer) of an existing structure and *name* is the local variable name (sometimes called the “instance name”). (See **STRUCT** on page V-678 for details.)

To access a member of a structure, specify the STRUCT variable name followed by a “.” and the member name:

```
STRUCT Point pt
pt.v= 100
```

When a member is defined as an array:

```
Structure mystruct
    Variable var1
    Variable var2 [10]
...
EndStructure
```

you must specify [*index*] to use a given element in the array:

```
STRUCT mystruct ms
ms.var2 [n] = 22
```

Structure and field names must be literal and can not use *\$str* notation. The *index* value can be calculated at runtime (it doesn’t have to be a literal number). If the field is itself a STRUCT, continue to append “.” and field names as needed. See the **Example** on page IV-80.

You can define an array of structures as a field in a structure:

```
Structure mystruct
    STRUCT Point pt [100]    // Allowed as a sub-structure
EndStructure
```

However, you can not define an array of structures as a local variable in a function:

```
STRUCT Point pt [100]    // Not allowed as a function local variable
```

WAVE, NVAR, SVAR, and FUNCREF members can be initialized using the same syntax as used for the corresponding nonstructure variables. See **Runtime Lookup of Globals** on page IV-50 and **Function References** on page IV-84.

## Chapter IV-3 — User-Defined Functions

---

Structures may be passed to functions **only** by reference (as with Fortran), which allows them to be both input and output parameters (see **Pass-By-Reference** on page IV-45). The syntax is:

```
STRUCT sName &varName
```

In a user function you define the input parameter:

```
Function myFunc(s)
    STRUCT mystruct &s
    ...
End
```

Char and uchar arrays can be treated as zero-terminated strings by leaving off the brackets. Because the Igor compiler knows the size, the entire array can be used with no zero termination. Like normal string variables, concatenation using += is allowed but substrings assignment using [p1, p2] = subStr is not supported.

Structures, including substructures, may be copied using simple assignment from one structure to the other. The source and destination structures must be defined using the same structure name (tag).

The **Print** operation (see page V-498) can print individual elements of a structure or can print a summary of the entire STRUCT variable.

### Example

Here is a (somewhat contrived) example using structures. Try executing `foo(2)`:

```
Constant kCaSize= 5

Structure substruct
    Variable v1
    Variable v2
EndStructure

Structure mystruct
    Variable var1
    Variable var2[10]
    String s1
    WAVE fred
    NVAR globVar1
    SVAR globStr1
    FUNCREF myDefaultFunc afunc
    STRUCT substruct ss1[3]
    char ca[kCaSize+1]
EndStructure

Function foo(n)
    Variable n

    Make/O/N=20 fred
    Variable/G globVar1= 111
    String/G aGlobStr="a global string var"

    STRUCT mystruct ms
    ms.var1= 11
    ms.var2[n]= 22
    ms.s1= "string s1"
    WAVE ms.fred // could have =name if want other than waves named fred
    NVAR ms.globVar1
    SVAR ms.globStr1= aGlobStr
    FUNCREF myDefaultFunc ms.afunc= anotherfunc
    ms.ss1[n].v1= ms.var1/2
    ms.ss1[0]= ms.ss1[n]
    ms.ca= "0123456789"
    bar(ms,n)
    print ms.var1,ms.var2[n],ms.s1,ms.globVar1,ms.globStr1,ms.ss1[n].v1
    print ms.ss1[n].v2,ms.ca,ms.afunc()
    print "a whole wave",ms.fred
    print "the whole ms struct:",ms

    STRUCT substruct ss
    ss= ms.ss1[n]
```

```

    print "copy of substruct",ss
End

Function bar(s,n)
    STRUCT mystruct &s
    Variable n

    s.ssl[n].v2= 99
    s.fred= sin(x)
    Display s.fred
End

Function myDefaultFunc()
    return 1
End

Function anotherfunc()
    return 2
End

```

Note the use of WAVE, NVAR, SVAR and FUNCREF in the function foo. These keywords are required both in the structure definition and again in the function, when the structure members are initialized.

### Built-In Structures

Igor includes a few special purpose, predefined structures for use with certain operations. Some of those structures use these predefined general purpose structures:

```

Structure Rect
    Int16 top, left, bottom, right
EndStructure

Structure Point
    Int16 v, h
EndStructure

Structure RGBColor
    UInt16 red, green, blue
EndStructure

```

A number of operations use built-in structures that the Igor programmer can use. See the command reference information for details about these structures and their members.

Operation	Structure Name
Button	WMButtonAction
CheckBox	WMCheckboxAction
CustomControl	WMCustomControlAction
ListBox	WMListboxAction
ModifyFreeAxis	WMAxisHookStruct
PopupMenu	WMPopupAction
SetVariable	WMSetVariableAction
SetWindow	WMWinHookStruct
Slider	WMSliderAction
TabControl	WMTabControlAction

### Applications of Structures

Structures are useful for reading and writing disk files. The **FBinRead** operation (see page V-153) and the **FBinWrite** operation (see page V-154) understand structure variables and will read or write the entire structure from or to a disk file. The individual fields of the structure will be byte-swapped if you use the /B flag.

Structures can be used in complex programming projects to reduce the dependency on global objects and to simplify passing data to and getting data from functions. For example, a base function might allocate a local structure variable and then pass that variable on to a large set of lower level routines. Because structure variables are passed by reference, data written into the structure by lower level routines is available to the higher level. Without structures, you would have to pass a large number of individual parameters or use global variables and data folders.

### Using Structures with Windows and Controls

Predefined structures are available for “new-style” (new as of Igor Pro 5) action procedures for controls and hook functions for windows. See the documentation for each of the individual control listed in Chapter III-14, **Controls and Control Panels** and in Chapter V-1, **Igor Reference**, and the **SetWindow** operation (see page V-569).

Advanced programmers should also be aware of userdata that can be associated with windows using the **SetWindow** operation (see page V-569). Userdata is binary data that persists with individual windows; it is suitable for storing structures. Storing structures in a window’s userdata is very handy in eliminating the need for global variables and reduces the bookkeeping needed to synchronize those globals with the window’s life cycle. Userdata is also available for use with controls. See the **ControlInfo** operation (see page V-63), **GetWindow** operation (see page V-225), **GetUserData** operation (see page V-224), and **SetWindow** operation (see page V-569) operations.

#### Example

Here is an example illustrating Igor- and user-defined structures along with userdata in a control. Put the following in the procedure window of a new experiment and run the Panel0 macro. Then click on the buttons. Note that the buttons will remember their state even if the experiment is saved and reloaded. To fully understand this example, examine the definition of WMButtonAction in the **Button** operation (see page V-38).

```
#pragma rtGlobals=1          // Use modern global access method.

Structure mystruct
    Int32 nclicks
    double lastTime
EndStructure

Function ButtonProc(bStruct) : ButtonControl
    STRUCT WMButtonAction &bStruct

    if( bStruct.eventCode != 1 )
        return 0          // we only handle mouse down
    endif

    STRUCT mystruct s1
    if( strlen(bStruct.userdata) == 0 )
        print "first click"
    else
        StructGet/S s1,bStruct.userdata
        String ctime= Secs2Date(s1.lastTime, 1)+" "+Secs2Time(s1.lastTime,1)
        // Warning: Next command is wrapped to fit on the page.
        printf "button %s clicked %d time(s), last click =
%s\r",bStruct.ctrlName,s1.nclicks,ctime
    endif
    s1.nclicks += 1
    s1.lastTime= datetime
    StructPut/S s1,bStruct.userdata
    return 0
End

Window Panel0() : Panel
    PauseUpdate; Silent 1          // building window...
    NewPanel /W=(150,50,493,133)
```

```

SetDrawLayer UserBack
Button b0,pos={12,8},size={50,20},proc=ButtonProc,title="Click"
Button b1,pos={65,8},size={50,20},proc=ButtonProc,title="Click"
Button b2,pos={119,8},size={50,20},proc=ButtonProc,title="Click"
Button b3,pos={172,8},size={50,20},proc=ButtonProc,title="Click"
Button b4,pos={226,8},size={50,20},proc=ButtonProc,title="Click"
EndMacro

```

## Limitations of Structures

Although structures can reduce the need for global variables, they do not eliminate them altogether. A structure variable, like all local variables in functions, disappears when its host function returns. In order to maintain state information, you will need to store and retrieve structure information using global variables. You can do this using a global variable for each field or, with certain restrictions, you can store entire structure variables in a single global using the **StructPut** operation (see page V-679) and the **StructGet** operation (see page V-679).

As of Igor Pro 5.03, a structure can be passed to an external operation or function. See the Igor XOP Toolkit manual for details.

## Static Functions

You can create functions that are local to the procedure file in which they appear by inserting the keyword *Static* in front of *Function* (see **Static** on page V-594 for usage details). The main reason for using this technique is to minimize the possibility of your function names conflicting with other names. You can use common intuitive names instead of devising a special and sometimes ugly naming strategy.

Functions normally have global scope and are available in any part of an Igor experiment, but the static keyword limits the scope of the function to its procedure file and hides it from all other procedure files. Static functions can only be used in the file in which they are defined. They can not be called from the command line and they cannot be accessed from macros.

Because static functions cannot be executed from the command line, you will have write a public test function to test them.

You can break this rule and access a static function using a module name (see **Regular Modules** on page IV-212). This technique can test the function from the command line.

Non-static functions (functions without the static keyword) are sometimes called “public” functions.

## ThreadSafe Functions

ThreadSafe user functions provide support for computers with multiple processors and can be used for preemptive multitasking background tasks. A ThreadSafe function is one that can operate correctly during simultaneous execution by multiple threads.

**Note:** Writing a multitasking program is for expert programmers only. Intermediate programmers can write thread-safe curve-fitting functions and multithreaded assignment statements (see **Automatic Parallel Processing with MultiThread** on page IV-283). Beginning programmers should gain experience with regular programming before using multitasking.

You can create thread safe functions by inserting the keyword **ThreadSafe** in front of **Function**. For example:

```

ThreadSafe Function myadd(a,b)
  Variable a,b

  return a+b
End

```

Only a subset of functions and operations can be used in a ThreadSafe function. Generally, all numeric or utility functions can be used but those that access windows can not. To determine if a routine is ThreadSafe, use the Command Help tab of the Help Browser.

## Chapter IV-3 — User-Defined Functions

---

Although file operations are listed as ThreadSafe, they have certain limitations when running in a ThreadSafe function. If a file load hits a condition that normally would need user assistance, the load is aborted. No printing to history is done and there is no support for symbolic paths (use **PathInfo** and pass the path as a string input parameter).

ThreadSafe functions can call other ThreadSafe functions but may not call non-ThreadSafe functions. Non-ThreadSafe functions can call ThreadSafe functions.

When ThreadSafe functions execute in the main thread, they have normal access to data folders, waves, and variables. But when running in a preemptive thread, ThreadSafe functions use their own private data folders, waves, and variables. When a thread is started, waves can be passed to the function as input parameters. Such waves are then flagged as being in use by the thread, which prevents any changes to the size of the wave. When all threads under a given main thread are finished, the waves return to normal. You can pass data folders between the main thread and preemptive threads but such data folders are never shared.

See **ThreadSafe Functions and Multitasking** on page IV-288 for a discussion of programming with preemptive multitasking threads.

## Function Overrides

In some rare cases, you may need to temporarily change an existing function. When that function is part of a package provided by someone else (or by WaveMetrics) it may be undesirable or difficult to edit the original function. By using the keyword “Override” in front of “Function” you can define a new function that will be used in place of another function of the same name that is defined in a *different and later* procedure file.

Although it is difficult to determine the order in which procedure files are compiled, the main procedure window is always first. Therefore, always define override functions in the main procedure file.

Although you can override static functions, you may run into a few difficulties. If there are multiple files with the same static function name, your override will affect all of them, and if the different functions have different parameters then you will get a link error.

Here is an example of the Override keyword. In this example, start with a new experiment and create a new procedure window. Insert the following in the new window (not the main procedure window).

```
Function foo()  
    print "this is foo"  
End
```

```
Function Test()  
    foo()  
End
```

Now, on the command line, execute Test(). You will see “this is foo” in the history.

Open the main procedure window and insert the following:

```
Override Function foo()  
    print "this is an override version of foo"  
End
```

Now execute Test() again. You will see the “this is an override version of foo” in the history.

## Function References

Function references provide a way to pass a function to a function. This is a technique for advanced programmers. If you are just starting with Igor programming, you may want to skip this section and come back to it later.

To specify that an input parameter to a function is a function reference, use the following syntax:

```
Function Example(f)
    FUNCREF myprotofunc f
    . . .
End
```

This specifies that the input parameter *f* is a function reference and that a function named *myprotofunc* specifies the kind of function that is legal to pass. The calling function passes the name of a function as the *f* parameter. The called function can use *f* just as it would use the prototype function.

If a valid function is not passed then the prototype function will be called instead. The prototype function can either be a default function or it can contain error handling code that makes it obvious that a proper function was not passed.

A similar syntax can be used to create function reference variables in the body of a function:

```
FUNCREF protoFunc f = funcName
FUNCREF protoFunc f = $"str"
FUNCREF protoFunc f = funcRef
```

As shown, the right hand side can take either a literal function name, a *\$* expression that evaluates to a function name at runtime, or it can take another FUNCREF variable.

FUNCREF variables can refer to external functions as well as user functions. However, the prototype function must be a user function and it must not be static.

Although you can store a reference to a static function in a FUNCREF variable, you can not then use that variable with Igor operations that take a function as an input. *FuncFit* is an example of such an operation.

Following are some example functions and FUNCREFs that illustrate several concepts:

```
Function myprotofunc(a)
    Variable a

    print "in myprotofunc with a= ",a
End
```

```
Function foo1(var1)
    Variable var1

    print "in foo1 with a= ",var1
End
```

```
Function foo2(a)
    Variable a

    print "in foo2 with a= ",a
End
```

```
Function foo3(a)
    Variable a

    print "in foo3 with a= ",a
End
```

```
Function badfoo(a,b)
    Variable a,b

    print "in badfoo with a= ",a
End
```

```
Function bar(a,b,fin)
    Variable a,b
    FUNCREF myprotofunc fin
```

```
if( a==1 )
  FUNCREF myprotofunc f= foo1
elseif( a==2 )
  FUNCREF myprotofunc f= $"foo2"
elseif( a==3 )
  FUNCREF myprotofunc f= fin
endif
f(b)
End
```

For the above functions, the following table shows the results for various invocations of the bar function executed on the command line:

<b>Command</b>	<b>Result</b>
bar(1,33,foo3)	in foo1 with a= 33
bar(2,44,foo3)	in foo2 with a= 44
bar(3,55,foo3)	in foo3 with a= 55
bar(4,55,foo3)	in myprotofunc with a= 55

Executing `bar(3,55,badfoo)` will generate a syntax error dialog that will highlight “badfoo” in the command. This error results because the format of the badfoo function does not match the format of the prototype function, myprotofunc.

## Conditional Compilation

Compiler directives can be used to conditionally include or exclude blocks of code. This is especially useful when an XOP may or may not be available. It is also convenient for testing and debugging code. For example, to enable a block of procedure code depending on the presence or absence of an XOP use

```
#if Exists("nameOfAnXopRoutine")
  <procedure code using XOP routines>
#endif
```

The conditional compiler directives are modeled after the C/C++ language. Unlike other #keyword directives, these may be indented. For defining symbols, the directives are:

```
#define symbol
#undef symbol
```

For conditional compilation, the directives are:

```
#ifdef symbol
#endif
#ifndef symbol
#endif
#if expression
#elif expression
#else
#endif
```

Expressions are ordinary Igor expressions, but cannot involve any user-defined objects. They evaluate to TRUE if the absolute value is > 0.5.

Conditionals must be either completely outside or completely inside function definitions; they cannot straddle a function definition. Conditionals cannot be used within macros but the **defined** function can.

Nesting depth is limited to 16 levels. Trailing text other than a comment is illegal.

Note that `#define` is used purely for defining symbols (there is nothing like C's preprocessor) and the only use of a symbol is with `#if`, `#ifdef`, `#ifndef` and the defined function.

The defined function was added in Igor Pro 6.20 allowing the use of:

```
#if defined(symbol)
```

Unlike C, you cannot use `#if defined(symbol)`.

Symbols exist only in the file where they are defined; the only exception is for symbols defined in the main procedure window, which are available to all other procedures except independent modules. In addition, you can define global symbols that are available in all procedure windows (including independent modules) using:

```
SetIgorOption poundDefine=symb
```

This adds one symbol to a global list. You can query the global list using:

```
SetIgorOption poundDefine=symb?
```

This sets `V_flag` to TRUE if `symb` exists. To remove a symbol from the global list use:

```
SetIgorOption poundUndefine=symb
```

For non-independent module procedure windows, a symbol is defined if it exists in the global list *or* in the main procedure window's list *or* in the given procedure window.

For independent module procedure windows, a symbol is defined if it exists in the global list *or* in the given procedure window; it does not use the main procedure window list.

A symbol defined in a global list is not undefined by a `#undef` in a procedure window.

### Predefined Global Symbols

As of Igor Pro 6.20, the following global symbols are automatically predefined as appropriate, and available in all procedure windows:

Symbol	Automatically Predefined If
MACINTOSH	The Igor application is a Macintosh application.
WINDOWS	The Igor application is a Windows application.
IGOR64	The Igor application is a 64-bit application.

### Conditional Compilation Examples

```
#define MYSYMBOL

#ifdef MYSYMBOL

Function foo()
    print "This is foo when MYSYMBOL is defined"
End

#else

Function foo()
    print "This is foo when MYSYMBOL is NOT defined"
End

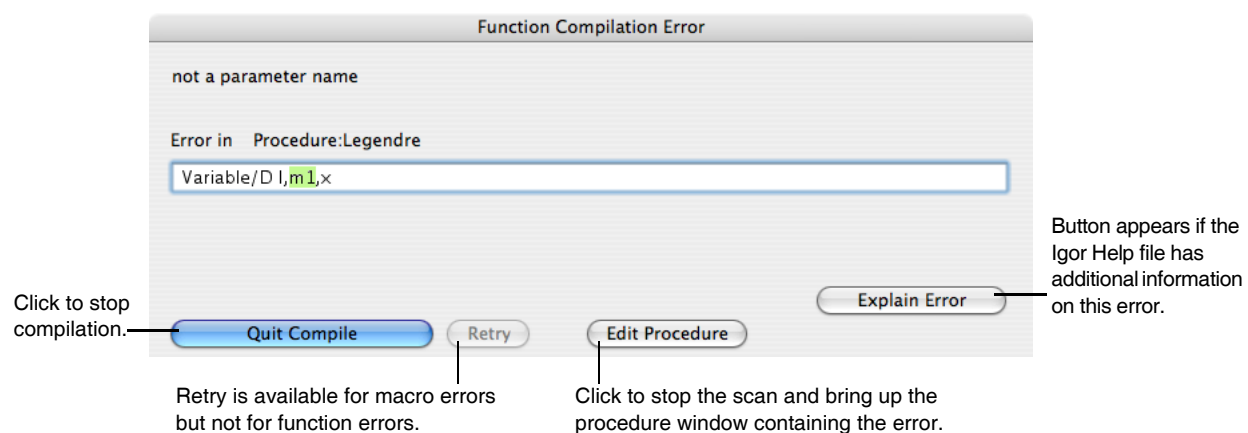
#endif    // MYSYMBOL

// This works in Igor Pro 6.20 or later
#ifdef MACINTOSH
    <conditionally compiled code here>
#endif
```

```
// This works in Igor Pro 6.00 or later
#if CmpStr("Macintosh",IgorInfo(2)) == 0
    <conditionally compiled code here>
#endif
```

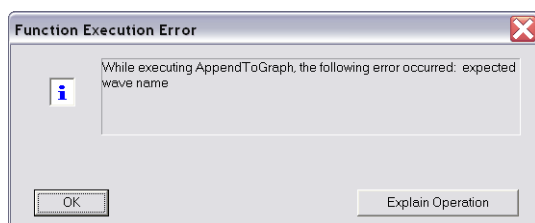
### Function Errors

During function compilation, Igor checks and reports syntactic errors and errors in parameter declarations. The normal course of action is to edit the offending function and try to compile again.



Runtime errors in functions are not reported on the spot. Instead, Igor saves information about the error and function execution continues. Igor presents an error dialog only after the last function ceases execution and Igor returns to the idle state. If multiple runtime errors occur, only the first is reported.

When a runtime error occurs, after function execution ends, Igor will present a dialog:



In this example, we tried to pass to the AppendToGraph function a reference to a wave that did not exist. To find the source of the error, you should use Igor's debugger and set it to break on error (see **Debugging on Error** on page IV-185 for details).

Sophisticated programmers may want to detect and deal with runtime errors on their own. The **GetRTError** function (see page V-221) can be used to check if an error occurred, and optionally to clear the error so that Igor doesn't report it.

### Coercion in Functions

The term "coercion" means the conversion of a value from one numeric precision or numeric type to another. Consider this example:

```
Function foo(awave)
    WAVE/C awave

    Variable/C var1
```

```

    var1= awave[2]*cplx(2,3)
    return real(var1)
End

```

The parameter declaration specifies that `awave` is complex. You can pass any kind of wave you like but it will be coerced into complex before use. For example, if you pass a real valued integer wave the value at point index 2 will be converted to double precision and zero will be used for the imaginary part.

## Operations in Functions

You can call most operations from user-defined functions. To provide this capability, WaveMetrics had to create special code for each operation. Some operations weren't worth the trouble or could cause problems. If an operation can't be invoked from a function, an error message is displayed when the function is compiled. The operations that can't be called from a function are:

<code>AppendToLayout</code>	<code>Layout</code>	<code>Modify</code>	<code>OpenProc</code>
<code>PrintGraphs</code>	<code>Quit</code>	<code>See Also</code>	<code>Stack</code>
<code>StackWindows</code>	<code>Tile</code>	<code>TileWindows</code>	

If you need to invoke one of these operations from a user-defined function, use the `Execute` operation. See **The Execute Operation** on page IV-176.

Note that the `ModifyGraph`, `ModifyTable` and `ModifyLayout` operations *can* be called from a function.

Also, you can use the **NewLayout** operation (see page V-437) instead of `Layout`, the **AppendLayoutObject** operation (see page V-23) instead of `AppendToLayout`, and the **RemoveLayoutObjects** operation (see page V-518) instead of `RemoveFromLayout`.

External operations implemented by old XOPs also can not be called directly from user-defined functions. Again, the solution is to use the `Execute` operation.

## Updates During Function Execution

An update is an action that Igor performs which consists of:

- Reexecuting formulas for dependent objects whose antecedents have changed (see Chapter IV-9, **Dependencies**);
- Redrawing graphs and tables which display waves that have changed;
- Redrawing page layouts containing graphs, tables, or annotations that have changed;
- Redrawing windows that have been uncovered.

When no procedure is executing, Igor continually checks whether an update is needed and does an update if necessary.

When a function is executing, Igor does no automatic updates at all. However, you can force an update by calling the **DoUpdate** operation (see page V-121). Call `DoUpdate` if you don't want to wait for the next automatic update which will occur when function execution finishes.

## Aborting Functions

There are two ways to prematurely stop procedure execution: a user abort or a programmed abort. Both stop execution of all procedures, no matter how deeply nested.

On Macintosh, you can abort procedure execution by pressing Command-period. On Windows, press Ctrl+Break or click the Abort button in the status bar. You may need to hold the keys down for a while because Igor looks at the keyboard periodically and if you don't press the keys long enough, Igor will not see them.

## Chapter IV-3 — User-Defined Functions

---

A user abort does not directly return. Instead it sets a flag that stops loops from looping and then returns using the normal calling chain. For this reason some code will still be executed after an abort but execution should stop quickly. This behavior releases any temporary memory allocations made during execution.

A **programmed abort** occurs during procedure execution according to conditions set by the programmer.

The simplest programmed abort occurs when the **Abort** operation (see page V-16) is executed. Here is an example:

```
if( numCells > 10 )
  Abort "Too many cells! Quitting."
endif
// code here doesn't execute if numCells > 10
```

Other programmed aborts can be triggered using the `AbortOnRTE` and `AbortOnValue` flow control keywords. The try-catch-endtry flow control construct can be used for catching and testing for aborts. See **Flow Control for Aborts** on page IV-38 for more details.

## Legacy Code Issues

This section discusses changes that have occurred in Igor programming over the years. If you are writing new code, you don't need to be concerned with these issues. If you are working with existing code, you may run into some of them.

If you are just starting to learn Igor programming, you have enough to think about already, so it is a good idea to skip this section. Once you are comfortable with the modern techniques described above, come back and learn about these antiquated techniques.

### Old-Style Comments and Compatibility Mode

As of Igor Pro 4.0, the old comment character, a vertical bar (`|`), has been replaced and should no longer be used when writing new procedures. In Igor Pro 4.0 and later, the `|` character is used as the bitwise OR operator (see **Operators** on page IV-5).

In order to run old procedures which use `|` as the comment symbol, Igor supports a compatibility mode. This mode is a property of an experiment. When the current experiment is in compatibility mode, Igor interprets `|` as a comment symbol. Normally, when the current experiment is not in compatibility mode, Igor interprets `|` as bitwise OR.

You put the current experiment in compatibility mode by executing the following on the command line:

```
Silent 100
```

You take the current experiment out of compatibility mode by executing:

```
Silent 101
```

When you execute these commands, Igor automatically recompiles all procedure files in the current experiment using the new mode.

All experiments created pre-Igor 4 are automatically in compatibility mode until you update them and their procedures.

We strongly recommend that you update old experiments and procedures so that you don't need to use compatibility mode. Until you do so, your old experiments will not work with new procedure files that use `|` as bitwise OR. Here are the steps to update an experiment and its procedures:

1. In each old procedure file, replace each `|` symbol that is used to introduce a comment with `//`. Use Edit→Find and Edit→Find Same to find each occurrence of `|`. Do not do a mass replace because you may inadvertently change a `|` symbol used in the old bitwise OR operator (`%|`, which is still supported) or in a string.
2. On the command line, take the experiment out of compatibility mode by executing:

```
Silent 101
```

Igor will recompile procedures.

3. If there are any remaining obsolete uses of |, Igor will display a compile error dialog. Fix the error and recompile the procedures until you get no more errors.

If you create procedure files that are used by other people (either in your group or for public use) and you want to use the new logic operations, such as | (bitwise OR) or || (logical OR), which require compatibility mode to be off (`Silent 101`), then you can specify

```
#pragma rtGlobals=2
```

in place of the normal `rtGlobals=1`.

If your procedure file is included in an experiment running in compatibility mode (`Silent 100`) then an alert dialog will be presented that will allow the user to turn compatibility mode off. However, keep in mind that when the procedures are recompiled in the new mode, the user's other procedures will generate errors if they use the obsolete comment symbol.

### Text After Flow Control

Prior to Igor Pro 4, Igor ignored any extraneous text after a flow control statement. Such text was an error, but Igor did not detect it.

Igor now checks for extra text after flow control statements. When found, a dialog is presented asking the user if such text should be considered an error or not. The answer lasts for the life of the Igor session.

Because previous versions of Igor ignored this extra text, it may be common for existing procedure files to have this problem. The text may in many cases simply be a typographic error such as an extra closing parenthesis:

```
if( a==1 )
```

In other cases, the programmer may have thought they were creating an elseif construct:

```
else if( a==2 )
```

even though the "if(a==2)" part was simply ignored. In some cases this may represent a bug in the programmer's code but most of the time it is asymptomatic.

### Global Variables Used by Igor Operations

The section **Local Variables Used by Igor Operations** on page IV-47 explains that certain Igor operations create and set certain special local variables. That is true if the procedure file uses runtime lookup of globals (`rtGlobals=1` is in effect). In very old procedure files that use the obsolete direct reference method of accessing variables, the operations create and set global variables.

Also explained in **Local Variables Used by Igor Operations** on page IV-47 is the fact that some operations, such as `CurveFit`, look for certain special local variables which modify the behavior of the operations. For historic reasons, operations that look for special variables will look for global variables in the current data folder if the local variable is not found. This is true whether `rtGlobals=1` is in effect or not. This behavior is unfortunate and may be removed from Igor some day. New programming should use local variables for this purpose.

### Direct Reference to Globals

The section **Accessing Global Variables and Waves** on page IV-50 explains how to access globals from a procedure file that uses runtime lookup of globals (`rtGlobals=1` is in effect). This section explains accessing globals in very old procedure files that use the obsolete direct reference method of accessing variables.

Prior to Igor Pro 3, Igor required that globals referenced from functions had to exist when the function was compiled. This was called the "direct reference" method of accessing globals. In Igor Pro 3, a new method, called the "runtime lookup" method, was added.

In a particular procedure file, the presence of the statement:

```
#pragma rtGlobals = 1
```

## Chapter IV-3 — User-Defined Functions

---

specifies that the file uses the runtime lookup method. If this statement is absent, or if the number used is 0 instead of the usual 1, the file uses the old direct reference method.

All new programming should use the runtime lookup method and consequently all new procedure files should contain the `rtGlobals` pragma as shown above.

Future versions of Igor may remove the old method. Therefore Igor programmers should modify old procedure files to use it.

This section explains the old method. It is of interest only if you need to work with old files.

To compile a direct reference to a global, the global must exist at compile time. There are three ways to meet this requirement:

1. Create the global from the command line before compiling the procedures.
2. Declare the global in a function before the first use of the global.
3. Let Igor automatically create the global.

Method 1 works for waves, numeric variables and string variables. However, methods 2 and 3 work for numeric and string variables only, not for waves.

To use method 1, execute a `Make`, `Variable`, or `String` command from the command line before compiling procedures that refer to the global. This technique is not good for use with a separate utility procedure file because you would need to remember to create the globals each time you opened the file.

To use method 2, create a function at the top of the procedure file that declares all of the global variables used in the procedure file. For example:

```
#pragma rtGlobals=0          // Use old direct reference method

Function InitFilterProcGlobals()
    Variable/G gCutoffFrequency, gFilterType
    String/G gFilterErrorMessage

    gCutoffFrequency = 1.0
    gFilterType = 0
    gFilterErrorMessage = ""
End
```

This method works because, with `rtGlobals=0`, Igor creates the declared global variables and strings at compile time. When Igor compiles the `Variable/G` and `String/G` declarations, it creates the globals right then and there. Thus, the globals are guaranteed to exist when referenced later in this function or in any other function compiled after this function.

Although the globals are created at compile time, they will have default values (0 for numeric globals, "" for string globals) until the function actually runs. Compile-time declaration creates the globals but does not set their initial values.

Method number 3 occurs automatically when you compile, with `rtGlobals=0`, an operation (e.g., `WaveStats`) that returns results via global variables. Therefore the following will compile without error:

```
#pragma rtGlobals=0          // Use old direct reference method

Function WaveStdDev(w)
    WAVE w

    WaveStats/Q w
    return V_sdev
End
```

`V_sdev` is one of many global variable created by the `WaveStats` operation at compile time (if `rtGlobals=0` is in effect) but set to a value at runtime. Thus, when you compile `WaveStats` in a function, Igor creates these global variables.

By default (if a given procedure file contains no `rtGlobals` statement), `rtGlobals=0` is in effect. The “`#pragma rtGlobals=1`” statement affects only the procedure file in which it appears. If you enter it in your main procedure window, it will not affect any other included or explicitly opened procedure files. Use this to modernize one procedure file at a time.

The `rtGlobals` pragma can appear anywhere in the procedure file, even inside a function, and it can appear more than once. However, it should be sufficient in nearly all cases to have just one `rtGlobals` pragma near the beginning of the file.

Here are the steps for converting a procedure file to use the runtime lookup method for accessing globals:

1. Insert the `#pragma rtGlobals=1` statement, with no indentation, at or near the top of the procedure in the file.
2. Click the **Compile** button to compile the procedures.
3. If you use direct reference to access a global, Igor will display an error dialog indicating the line on which the error occurred. Add an `NVAR`, `SVAR` or `WAVE` reference.
4. If you encountered an error in step 3, return to step 2.

