Chapter
IV-7

Programming Techniques

Overview

This chapter discusses programming techniques and issues that go beyond the basics of the Igor programming language and which all Igor programmers should understand. Techniques for advanced programmers are discussed in Chapter IV-10, **Advanced Topics**.

The Include Statement

The include statement is a compiler directive that you can put in your procedure file to open another procedure file. A typical include statement looks like this:

```
#include <Split Axis>
```

This statement opens the file "Split Axis.ipf" in the WaveMetrics Procedures folder. Once this statement has been compiled, you can call routines from that file. This is the recommended way to access procedure files that contain utility routines.

The # character at the beginning specifies that a compiler directive follows. Compiler directives, except for conditional compilation directives, must start at the far left edge of the procedure window with no leading tabs or spaces. Include statements can appear anywhere in the file but it is conventional to put them near the top.

It is possible to include a file which in turn includes other files. Igor opens all of the included files.

Igor opens included files when it compiles procedures. If you remove an include statement from a procedure file, Igor automatically closes the file on the next compile.

There are four forms of the include statement:

1. Igor searches for the named file in "Igor Pro 7 Folder/WaveMetrics Procedures" and in any subfolders:

   `#include <`*`fileName`*`>`

   Example: `#include <Split Axis>`
2. Igor searches for the named file in "Igor Pro 7 Folder/User Procedures" and "Igor Pro User Files/User Procedures" and in any subfolders:

   `#include "`*`fileName`*`"`

   Example: `#include "Utility Procs"`
3. Igor looks for the file only in the exact location specified:

   `#include "`*`full file path`*`"`

   Example: `#include "Hard Disk:Procedures:Utility Procs"`
4. Igor looks for the file relative to the Igor Pro 7 folder and the Igor Pro User Files folder and the folder containing the procedure file that contains the #include statement:

   `#include "`*`:partial file path`*`"`

   Example: `#include ":Spectroscopy Procedures:Voigt Procs"`

   Igor looks first relative to the Igor Pro 7 Folder. If that fails, it looks relative to the **Igor Pro User Files** folder. If that fails it looks relative to the procedure file containing the #include statement or, if the #include statement is in the built-in procedure window, relative to the experiment file.

   The name of the file being included must end with the standard ".ipf" extension but the extension is not used in the include statement.

Procedure File Version Information

If you create a procedure file to be used by other Igor users, it's a good idea to add version information to the file. You do this by putting a #pragma version statement in the procedure file. For example:

```
#pragma version = 1.10
```

This statement must appear with no indentation and must appear in the first 50 lines of the file. If Igor finds no version pragma, it treats the file as version 1.00.

Igor looks for the version information when the user invokes the File Information dialog from the Procedure menu. If the file has version information, Igor displays the version next to the file name in the dialog.

Igor also looks for version information when it opens an included file. An include statement can require a certain version of the included file using the following syntax:

```
#include <Bivariate Histogram> version>=1.02
```

If the required version of the procedure file is not available, Igor displays a warning to inform the user that the procedure file needs to be updated.

Turning the Included File's Menus Off

Normally an included procedure file's menus and menu items are displayed. However, if you are including a file merely to call one of its procedures, you may not want that file's menu items to appear. To suppress the file's menus and menu items, use:

```
#include <Decimation> menus=0
```

To use both the menus and version options, you must separate them with a comma:

```
#include <Decimation> menus=0, version>=1.1
```

Optionally Including Files

Compilation usually ceases and returns an error if the included file isn't found. On occasion it is advantageous to allow compilation to proceed if an included file isn't present or is the wrong version. Optionally including a procedure file is appropriate only if the file truly isn't needed to make procedures compile or operate.

Use the "optional" keyword to optionally include a procedure file:

```
#include <Decimation> version>=1.1, optional
```

Writing General-Purpose Procedures

Procedures can be placed on a scale ranging from general to specific. Usually, the high-level procedures of a program are specific to the task at hand and call on more general, lower-level procedures. The most general procedures are often called "utility" procedures.

You can achieve a high degree of productivity by building a library of utility procedures that you reuse in different programs. In Igor, you can think of the routines in an experiment's built-in Procedure window as a program. It can call utility routines which should be stored in a separate procedure file so that they are available to multiple experiments.

The files stored in the WaveMetrics Procedures folder contain general-purpose procedures on which your high-level procedures can build. Use the include statement (see **The Include Statement** on page IV-155) to access these and other utility files.

When you write utility routines, you should keep them as general as possible so that they can be reused as much as possible. Here are some guidelines.

- Avoid the use of global variables as inputs or outputs. Using globals hard-wires the routine to specific names which makes it difficult to use and limits its generality.
- If you use globals to store state information between invocations of a routine, package the globals in data folders.

  WaveMetrics packages usually create data folders inside "root:Packages". We use the prefix "WM" for data folder names to avoid conflict with other packages. Follow this lead and should pick your own data folder names so as to minimize the chance of conflict with WaveMetrics packages or with others. See **Managing Package Data** on page IV-234 for details.
- Choose a clear and specific name for your utility routine.

By choosing a name that says precisely what your utility routine does, you minimize the likelihood of collision with the name of another procedure. You also increase the readability of your program.

- Make functions which are used only internally by your procedures static.

  By making internal functions static (i.e., private), you minimize the likelihood of collision with the name of another procedure.

Programming with Liberal Names

Standard names in Igor can contain letters, numbers and the underscore character only.

It is possible to use wave names and data folder names that contain almost any character. Such names are called "liberal names" (see **Liberal Object Names** on page III-444).

As this section explains, programmers need to use special techniques for their procedures to work with liberal names. Consequently, if you do use liberal names, some existing Igor procedures may break.

Whenever a liberal name is used in a command or expression, the name must be enclosed in single quotes. For example:

```
Make 'Wave 1', wave2, 'miles/hour'
'Wave 1' = wave2 + 'miles/hour'
```

Without the single quotes, Igor has no way to know where a particular name ends. This is a problem whenever Igor parses a command or statement. Igor parses commands at the following times:

- When it compiles a user-defined function.
- When it compiles the right-hand side of an assignment statement, including a formula (:= dependency expression).
- When it interprets a macro.
- When it interprets a command that you enter in the command line or via an Igor Text file.
- When it interprets a command you submit for execution via the Execute operation.
- When it interprets a command that an XOP submits for execution via the XOPCommand or XOPSilentCommand callback routines.

When you use an Igor dialog to generate a command, Igor automatically uses quotes where necessary.

Programmers need to be concerned about liberal names whenever they create a command and then execute it, via an Igor Text file, the Execute operation or the XOPCommand and XOPSilentCommand callback routines, and when creating a formula. In short, when you create something that Igor has to parse, names must be quoted if they are liberal. Names that are not liberal can be quoted or unquoted.

If you have a procedure that builds up a command in a string variable and then executes it via the Execute operation, you must use the PossiblyQuoteName function to provide the quotes if needed.

Here is a trivial example showing the liberal-name unaware and liberal-name aware techniques:

```
Function AddOneToWave(w)
   WAVE w

   String cmd
   String name = NameOfWave(w)

   // Liberal-name unaware - generates error with liberal name
   sprintf cmd, "%s += 1", name
   Execute cmd

   // Liberal-name aware way
   sprintf cmd, "%s += 1", PossiblyQuoteName(name)
   Execute cmd
End
```

Imagine that you pass a wave named *wave 1* to this function. Using the old technique, the Execute operation would see the following text:

```
wave 1 += 1
```

Igor would generate an error because it would try to find an operation, macro, function, wave or variable named *wave*.

Using the new technique, the Execute operation would see the following text:

```
'wave 1' += 1
```

This works correctly because Igor sees 'wave 1' as a single name.

When you pass a string expression containing a simple name to the $ operator, the name must not be quoted because Igor does no parsing:

```
String name = "wave 1"; Display $name                // Right
String name = "'wave 1'"; Display $name              // Wrong
```

However, when you pass a *path* in a string to $, any liberal names in the path must be quoted:

```
String path = "root:'My Data':'wave 1'"; Display $path// Right
String path = "root:My Data:wave 1"; Display $path    // Wrong
```

For further explanation of liberal name quoting rules, see **Accessing Global Variables and Waves Using Liberal Names** on page IV-62.

Some built-in string functions return a list of waves. WaveList is the most common example. If liberal wave names are used, you will have to be extra careful when using the list of names. For example, you can't just append the list to the name of an operation that takes a list of names and then execute the resulting string. Rather, you will have to extract each name in turn, possibly quote it and then append it to the operation.

For example, the following will work if all wave names are standard but not if any wave names are liberal:

```
Execute "Display " + WaveList("*", ",", "")
```

Now you will need a string function that extracts out each name, possibly quotes it, and creates a new string using the new names separated by commas. The PossiblyQuoteList function in the WaveMetrics procedure file Strings as Lists does this. The preceding command becomes:

```
Execute "Display " + PossiblyQuoteList(WaveList("*", ",", ""), ",")
```

To include the Strings as Lists file in your procedures, see **The Include Statement** on page IV-155.

For details on using liberal names in user-defined functions, see **Accessing Global Variables and Waves Using Liberal Names** on page IV-62.

Programming with Data Folders

For general information on data folders, including syntax for using data folders in command line operations, see Chapter II-8, **Data Folders**.

Data folders provide a powerful way to organize data and reduce clutter. However, using data folders introduces some complexity in Igor programming. The name of a variable or wave is no longer sufficient to uniquely identify it because the name alone does not indicate in which data folder it resides.

If a procedure is designed to work on the current data folder, then it is the user's responsibility to set the current data folder correctly before running the procedure. When writing such a procedure, you can use **WaveExists**, **NVAR_Exists**, and **SVAR_Exists** to check the existence of expected objects and fail gracefully if they do not exist.

Data Folder Applications

There are two main uses for data folders:

- To store globals used by procedures to keep track of their state
- To store runs of data with identical structures

The following sections explore these applications.

Storing Procedure Globals

If you create packages of procedures, for example data acquisition, data analysis, or a specialized plot type, you typically need to store global variables, strings and waves to maintain the state of your package. You should keep all such items in a data folder that you create with a unique name to reduce clutter and avoid conflicts with other packages.

If your package is inherently global in nature, data acquisition for example, create your data folder within a data folder named Packages that you create in the root data folder. If your package is named MyDataAcq, you would store your globals in root:Packages:MyDataAcq. See **Managing Package Data** on page IV-234 for details and examples.

On the other hand, if your package needs to create a group of variables and waves as a result of an operation on a single input wave, it makes sense to create your data folder in the one that holds the input wave (see the **GetWavesDataFolder** function on page V-273). Similarly, if you create a package that takes an entire data folder as input (via a DFREF parameter pointing to the data folder) and needs to create a data folder containing output, it should create the output data folder in the one, or possibly at the same level as the one containing the input.

If your package creates and maintains windows, it should create a master data folder in root:Packages and then create individual data folders within the master that correspond to each instance of a window. For example, a Smith Chart package would create a master data folder as root:Packages:SmithCharts: and then create individual data folders inside the master with names that correspond to the individual graphs.

A package designed to operate on traces within graphs would go one step further and create a data folder for each trace that it has worked on, inside the data folder for the graph. This technique is illustrated by the Smooth Control Panel procedure file. For a demonstration, choose File→Examples→Feature Demos→Smoothing Control Panel.

Storing Runs of Data

If you acquire data using a well-established experimental protocol, your data will have a well-defined structure. Each time you run your protocol, you produce a new data set with the same structure. Storing each data set in its own data folder avoids name conflicts between corresponding items of different data sets. It also simplifies the writing of procedures to analyze and compare data set.

To use a trivial example, your data might have a structure like this:

```
<Run of data>
   String: conditions
   Variable: temperature
   Wave: appliedVoltage
   Wave: luminosity
```

Having defined this structure, you could then write procedures to:

1. Load your data into a data folder
2. Create a graph showing the data from one run
3. Create a graph comparing the data from many runs

Writing these procedures is greatly simplified by the fact that the names of the items in a run are fixed. For example, step 2 could be written as:

```
Function GraphOneRun()  // Graphs data run in the current data folder.
   SVAR conditions
   NVAR temperature
   WAVE appliedVoltage, luminosity

   Display luminosity vs appliedVoltage

   String text
   sprintf text, "Temperature: %g.\rExperimental conditions: %s.",
         temperature, conditions
   Textbox text
End
```

To create a graph, you would first set the current data folder to point to a run of data and then you would invoke the GraphOneRun function.

The Data Folder Tutorial experiment shows in detail how to accomplish the three steps listed above. Choose File→Example Experiments→Tutorials→Data Folder Tutuorial.

Setting and Restoring the Current Data Folder

There are many reasons why you might want to save and restore the current data folder. In this example, we have a function that does a curve fit to a wave passed in as a parameter. The CurveFit operation creates two waves, W_coef and W_sigma, in the current data folder. If you use the /D option, it also creates a destination wave in the current data folder. In this function, we make sure that the W_coef, W_sigma and destination waves are all created in the same data folder as the source wave.

```
Function DoLineFit(w)
   WAVE w

   String saveDF = GetDataFolder(1)
   SetDataFolder GetWavesDataFolder(w,1)
   CurveFit line w /D
   SetDataFolder saveDF
End
```

Many other operations create output waves in the current data folder. Depending on what your goal is, you may want to use the technique shown here to control where the output waves are created.

A function should always save and restore the current data folder unless it is designed explicitly to change the current data folder.

Determining a Function's Target Data Folder

There are three common methods for determining the data folder that a function works on or in:

1. Passing a wave in the target data folder as a parameter
2. Having the function work on the current data folder
3. Passing a DFREF parameter that points to the target data folder

For functions that operate on a specific wave, method 1 is appropriate.

For functions that operation on a large number of variables within a single data folder, methods 2 or 3 are appropriate. In method 2, the calling routine sets the data folder of interest as the current data folder. In method 3, the called function does this, and restores the original current data folder before it returns.

Clearing a Data Folder

There are times when you might want to clear a data folder before running a procedure, to remove things left over from a preceding run. If the data folder contains no child data folders, you can achieve this with:

```
KillWaves/A/Z; KillVariables/A/Z
```

If the data folder does contain child data folders, you could use the KillDataFolder operation. This operation kills a data folder and its contents, including any child data folders. You could kill the main data folder and then recreate it. A problem with this is that, if the data folder or its children contain a wave that is in use, you will generate an error which will cause your function to abort.

Here is a handy function that kills the contents of a data folder and the contents of its children without killing any data folders and without attempting to kill any waves that may be in use.

```
Function ZapDataInFolderTree(path)
   String path

   String saveDF = GetDataFolder(1)
   SetDataFolder path

   KillWaves/A/Z
   KillVariables/A/Z
   KillStrings/A/Z

   Variable i
   Variable numDataFolders = CountObjects(":", 4)
   for(i=0; i<numDataFolders; i+=1)
      String nextPath = GetIndexedObjName(":", 4, i)
      ZapDataInFolderTree(nextPath)
   endfor

   SetDataFolder saveDF
End
```

Using Strings

This section explains some common ways in which Igor procedures use strings. The most common techniques use built-in functions such as StringFromList and FindListItem. In addition to the built-in functions, there are a number of handy Igor procedure files in the WaveMetrics Procedures:Utilities:String Utilities folder.

Using Strings as Lists

Procedures often need to deal with lists of items. Such lists are usually represented as semicolon-separated text strings. The StringFromList function is used to extract each item, often in a loop. For example:

```
Function Test()
   Make jack,sam,fred,sue
   String list = WaveList("*",";","")
   Print list

   Variable numItems = ItemsInList(list), i
   for(i=0; i<numItems; i+=1)
      Print StringFromList(i,list)     // Print ith item
   endfor
End
```

For lists with a large number of items, using StringFromList as shown above is slow. This is because it must search from the start of the list to the desired item each time it is called. In Igor7 or later, you can iterate quickly using the optional StringFromList offset parameter:

```
Function Test()
   Make jack,sam,fred,sue
   String list = WaveList("*",";","")
   Print list

   Variable numItems = ItemsInList(list), i
   Variable offset = 0
   for(i=0; i<numItems; i+=1)
```

```
      String item = StringFromList(0, list, ";", offset) // Get ith item
      Print item
      offset += strlen(item) + 1
   endfor
End
```

See **ListToTextWave** for another fast way to iterate through the items in a large string list.

Using Keyword-Value Packed Strings

A collection of disparate values is often stored in a list of keyword-value pairs. For example, the TraceInfo and AxisInfo functions return such a list. The information in these strings follows this form

```
KEYWORD:value;KEYWORD:value; ...
```

The **keyword** is always upper case and followed by a colon. Then comes the **value** and a semicolon. When parsing such a string, you should avoid reliance on the specific ordering of keywords — the order is likely to change in future releases of Igor.

Two functions will extract information from keyword-value strings. NumberByKey is used when the data is numeric, and StringByKey is used when the information is in the form of text.

Using Strings with Extractable Commands

The **AxisInfo** and **TraceInfo** readback functions provide much of their information in a form that resembles the commands that you use to make the settings in the first place. For example, to set an axis to log mode, you would execute something like this:

```
ModifyGraph log(left)=1
```

If we now look at the characters of the string returned by AxisInfo we see:

```
AXTYPE:left;CWAVE:jack; ... RECREATION:grid(x)=0;log(x)=1 ...
```

To use this information, we need to extract the value following "log(x)=" and then use it in a ModifyGraph command with the extracted value with the desired graph as the target. Here is a user function that sets the log mode of the bottom axis to the same value as the left axis:

```
Function CopyLogMode()
   String info,text
   Variable pos
   info=AxisInfo("","left")
   pos= StrSearch(info,"RECREATION:",0) // skip to start of "RECREATION:"
   pos += strlen("RECREATION:")         // skip "RECREATION:", too
                                        // get text after "log(x)="
   text= StringByKey("log(x)",info[pos,inf],"=",";")

   String cmd = "ModifyGraph log(bottom)=" + text
   Execute cmd
End
```

Character-by-Character Operations

Igor functions like **strlen** and **strsearch**, as well as Igor string indexing (see **String Indexing** on page IV-15), are byte-oriented. That is, strlen returns the number of bytes in a string. Strsearch takes a byte position as a parameter. Expressions like `str[<index>]` return a single byte.

In Igor6, for western languages, each character is represented by a single byte, so the distinction between bytes and characters is insignificant, and byte-by-byte operations are sufficient.

In Igor7 and later, which use the UTF-8 text encoding to represent text in strings, all ASCII characters are represented by one byte but all non-ASCII characters are represented by multiple bytes (see **Text Encodings** on page III-409 for details). Consequently, byte-by-byte operations are insufficient for stepping through characters. Character-by-character operations are needed.

There is no straightforward way in Igor7 to step through text containing non-ASCII characters. In the rare cases where this is necessary, you can use the user-defined functions shown in this section.

The following functions show how to step through UTF-8 text character-by-character rather than byte-by-byte.

```
// NumBytesInUTF8Character(str, byteOffset)
// Returns the number of bytes in the UTF-8 character that starts byteOffset
// bytes from the start of str.
// NOTE: If byteOffset is invalid this routine returns 0.
//       Also, if str is not valid UTF-8 text, this routine return 1.
Function NumBytesInUTF8Character(str, byteOffset)
   String str
   Variable byteOffset

   Variable numBytesInString = strlen(str)
   if (byteOffset<0 || byteOffset>=numBytesInString)
      return 0          // Programmer error
   endif

   Variable firstByte = char2num(str[byteOffset]) & 0x00FF

   if (firstByte < 0x80)
      return 1
   endif

   if (firstByte>=0xC2 && firstByte<=0xDF)
      return 2
   endif

   if (firstByte>=0xE0 && firstByte<=0xEF)
      return 3
   endif

   if (firstByte>=0xF0 && firstByte<=0xF4)
      return 4
   endif

   // If we are here, str is not valid UTF-8.
   // Treat the first byte as a 1-byte character.
   // This prevents infinite loops.
   return 1
End

Function UTF8CharactersInString(str)
   String str

   Variable numCharacters = 0

   Variable length = strlen(str)
   Variable byteOffset = 0
   do
      if (byteOffset >= length)
         break
      endif
      Variable numBytesInCharacter = NumBytesInUTF8Character(str, byteOffset)
      if (numBytesInCharacter <= 0)
         Abort "Bug in CharactersInUTF8String"     // Should not happen
      endif
      numCharacters += 1
      byteOffset += numBytesInCharacter
   while(1)

   return numCharacters
```

```
End

Function/S UTF8CharacterAtPosition(str, charPos)
   String str
   Variable charPos

   if (charPos < 0)
      // Print "charPos is invalid"             // For debugging only
      return ""
   endif

   Variable length = strlen(str)
   Variable byteOffset = 0
   do
      if (byteOffset >= length)
         // Print "charPos is out-of-range"     // For debugging only
         return ""
      endif
      if (charPos == 0)
         break
      endif
      Variable numBytesInCharacter = NumBytesInUTF8Character(str, byteOffset)
      byteOffset += numBytesInCharacter
      charPos -= 1
   while(1)

   numBytesInCharacter = NumBytesInUTF8Character(str, byteOffset)
   String result = str[byteOffset, byteOffset+numBytesInCharacter-1]
   return result
End

Function DemoCharacterByCharacter()
   String str = "<string containing non-ASCII text>"

   Variable numCharacters = UTF8CharactersInString(str)
   Variable charPos

   for(charPos=0; charPos<numCharacters; charPos+=1)
      String character = UTF8CharacterAtPosition(str, charPos);
      Printf "CharPos=%d, char=\"%s\"\r", charPos, character
   endfor
End
```

See Also: **Text Encodings** on page III-409, **String Indexing** on page IV-15

Regular Expressions

A regular expression is a pattern that is matched against a subject string from left to right. Regular expressions are used to identify lines of text containing a particular pattern and to extracts substrings matching a particular pattern.

A regular expression can contain regular characters that match the same character in the subject and special characters, called "metacharacters", that match any character, a list of specific characters, or otherwise identify patterns.

The regular expression syntax is based on PCRE — the Perl-Compatible Regular Expression Library.

Igor syntax is similar to regular expressions supported by various UNIX and POSIX `egrep(1)` commands.

See **Regular Expressions References** on page IV-183 for details on PCRE.

Regular Expression Operations and Functions

Here are the Igor operations and functions that work with regular expressions:

Grep

The **Grep** operation identifies lines of text that match a pattern.

The subject is each line of a file or each row of a text wave or each line of the text in the clipboard.

Output is stored in a file or in a text wave or in the clipboard.

Grep Example

```
Function DemoGrep()
   Make/T source={"Monday","Tuesday","Wednesday","Thursday","Friday"}
   Make/T/N=0 dest
   Grep/E="sday" source as dest           // Find rows containing "sday"
   Print dest
End
```

The output from Print is:

```
   dest[0]= {"Tuesday","Wednesday","Thursday"}
```

GrepList

The **GrepList** function identifies items that match a pattern in a string containing a delimited list.

The subject is each item in the input list.

The output is a delimited list returned as the function result.

GrepList Example

```
Function DemoGrepList()
   String source = "Monday;Tuesday;Wednesday;Thursday;Friday"
   String dest = GrepList(source, "sday") // Find items containing "sday"
   Print dest
End
```

The output from Print is:

```
   Tuesday;Wednesday;Thursday;
```

GrepString

The **GrepString** function determines if a particular pattern exists in the input string.

The subject is the input string.

The output is 1 if the input string contains the pattern or 0 if not.

GrepString Example

```
Function DemoGrepString()
   String subject = "123.45"
   String regExp = "[0-9]+"            // Match one or more digits
   Print GrepString(subject,regExp)    // True if subject contains digit(s)
End
```

The output from Print is: 1

SplitString

The **Demo** operation identifies subpatterns in the input string.

The subject is the input string.

Output is stored in one or more string output variables.

SplitString Example

```
Function DemoSplitString()
   String subject = "Thursday, May 7, 2009"
   String regExp = "([[:alpha:]]+), ([[:alpha:]]+) ([[:digit:]]+), ([[:digit:]]+)"
   String dayOfWeek, month, dayOfMonth, year
   SplitString /E=(regExp) subject, dayOfWeek, month, dayOfMonth, year
   Print dayOfWeek, month, dayOfMonth, year
End
```

The output from Print is:

```
   Thursday  May  7  2009
```

Basic Regular Expressions

Here is a **Grep** command that uses "Fred" as the regular expression. "Fred" contains no metacharacters so this command identifies lines of text containing the literal string "Fred". It examines each line from the input file, afile.txt. All lines containing the pattern "Fred" are written to the output file, "FredFile.txt":

```
Grep/P=myPath/E="Fred" "afile.txt" as "FredFile.txt"
```

Character matching is case-sensitive by default. Prepending the Perl 5 modifier (?i) gives a case-insensitive pattern that matches upper and lower-case versions of “Fred”:

```
// Copy lines that contain "Fred", "fred", "FRED", "fREd", etc
Grep/P=myPath/E="(?i)fred" "afile.txt" as "AnyFredFile.txt"
```

To copy lines that do not match the regular expression, use the Grep /E flag with the optional *reverse* parameter set to 1 to reverse the sense of the match:

```
// Copy lines that do NOT contain "Fred", "fred", "fREd", etc
Grep/P=myPath/E={"(?i)fred",1} "afile.txt" as "NotFredFile.txt"
```

Note: Igor doesn't use the Perl opening and closing regular expression delimiter character which is the forward slash. In Perl you would use `"/Fred/"` and `"/(?i)fred/"`.

Regular Expression Metacharacters

The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of “metacharacters”, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of metacharacters: those that are recognized anywhere in the pattern except within brackets, and those that are recognized in brackets. Outside brackets, the metacharacters are as follows:

\	General escape character with several uses
^	Match start of string
$	Match end of string
.	Match any character except newline (by default) (To match newline, see **Matching Newlines** on page IV-173)
[	Start character class definition (for matching one of a set of characters)
|	Start of alternative branch (for matching one or the other of two patterns)
(	Start subpattern
)	End subpattern
?	0 or 1 quantifier (for matching 0 or 1 occurrence of a pattern) Also extends the meaning of (Also quantifier minimizer
*	0 or more quantifier (for matching 0 or more occurrence of a pattern)
+	1 or more quantifier (for matching 1 or more occurrence of a pattern) Also possessive quantifier
{	Start min/max quantifier (for matching a number or range of occurrences)

Character Classes in Regular Expressions

A character class is a set of characters and is used to specify that one character of the set should be matched. Character classes are introduced by a left-bracket and terminated by a right-bracket. For example:

```
[abc]       Matches a or b or c
[A-Z]       Matches any character from A to Z
[A-Za-z]    Matches any character from A to Z or a to z.
```

POSIX character classes specify the characters to be matched symbolically. For example:

```
[[:alpha:]] Matches any alphabetic character (A to Z or a to z).
[[:digit:]] Matches 0 to 9.
```

In a character class the only metacharacters are:

\	General escape character
^	Negate the class, but only if ^ is the first character
-	Indicates character range
[	POSIX character class (only if followed by POSIX syntax)
]	Terminates the character class

Backslash in Regular Expressions

The backslash character has several uses. First, if it is followed by a nonalphanumeric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, the * character normally means "match zero or more of the preceding subpattern". If you want to match a * character, you write * in the pattern. This escaping action applies whether or not the following character would otherwise be interpreted as a metacharacter, so it is always safe to precede a nonalphanumeric with backslash to specify that it stands for itself. In particular, if you want to match a backslash, you write \\.

Note: Because Igor also has special uses for backslash (see **Escape Sequences in Strings** on page IV-13), you must double the number of backslashes you would normally use for a Perl or grep pattern. Each pair of backslashes sends one backslash to, say, the **Grep** command.

For example, to copy lines that contain a backslash followed by a *z* character, the Perl pattern would be "\\z", but the equivalent Igor Grep expression would be /E="\\\\z".

Igor's input string parser converts "\\" to "\" so, when you write /E="\\\\z", the regular expression engine sees /E="\\z".

This difference is important enough that the PCRE and Igor Patterns (using Grep /E syntax) are both shown below when they differ.

Only ASCII numbers and letters have any special meaning after a backslash. All other characters are treated as literals.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between \Q and \E. This is different from Perl in that $ and @ are handled as literals in \Q...\E sequences in PCRE, whereas in Perl, $ and @ cause variable interpolation. Note the following examples:

Igor Pattern	PCRE Pattern	PCRE Matches	Perl Matches
\\Qabc$xyz\\E	\Qabc$xyz\E	abc$xyz	abc followed by the contents of $xyz
\\Qabc\\$xyz\\E	\Qabc\$xyz\E	abc\$xyz	abc\$xyz
\\Qabc\\E\\$\\Qxyz\\E	\Qabc\E\$\Qxyz\E	abc$xyz	abc$xyz

The \Q...\E sequence is recognized both inside and outside character classes.

Backslash and Nonprinting Characters

A second use of backslash provides a way of encoding nonprinting characters in patterns in a visible manner. There is no restriction on where nonprinting characters can occur, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

Igor Pattern	PCRE Pattern	Character Matched
\\a	\a	Alarm, that is, the BEL character (hex 07)
\\cx	\cx	“Control-x”, where x is any character
\\e	\e	Escape (hex 1B)
\\f	\f	Formfeed (hex 0C)
\\n	\n	Newline (hex 0A)
\\r	\r	Carriage return (hex 0D)
\\t	\t	Tab (hex 09)
\\ddd	\ddd	Character with octal code ddd, or backreference
\\xhh	\xhh	Character with hex code hh

Backslash and Nonprinting Characters Arcania

The material in this section is arcane and rarely needed. We recommend that you skip it.

The precise effect of \cx is if x is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus \cz becomes hex 1A, but \c{ becomes hex 3B, while \c; becomes hex 7B.

After \x, from zero to two hexadecimal digits are read (letters can be in upper or lower case). If characters other than hexadecimal digits appear between \x{ and }, or if there is no terminating }, this form of escape is not recognized. Instead, the initial \x will be interpreted as a basic hexadecimal escape, with no following digits, giving a character whose value is zero.

After \0 up to two further octal digits are read. In both cases, if there are fewer than two digits, just those that are present are used. Thus the sequence \0\x\07 specifies two binary zeros followed by a BEL character (code value 7). Make sure you supply two digits after the initial zero if the pattern character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a back reference. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE rereads up to three octal digits following the backslash, and generates a single byte from the least significant 8 bits of the value. Any subsequent digits stand for themselves. For example:

Igor Pattern	PCRE Pattern	Character(s) Matched
\\040	\040	Another way of writing a space
\\40	\40	A space, provided there are fewer than 40 previous capturing subpatterns
\\7	\7	Always a back reference
\\11	\11	Might be a back reference, or another way of writing a tab
\\011	\011	Always a tab
\\0113	\0113	A tab followed by the character “3”
\\113	\113	Might be a back reference, otherwise the character with octal code 113
\\377	\377	Might be a back reference, otherwise the byte consisting entirely of 1 bits
\\81	\81	Either a back reference or a binary zero followed by the two characters “8” and “1”

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single byte value can be used both inside and outside character classes. In addition, inside a character class, the sequence \b is interpreted as the backspace character (hex 08), and the sequence \X is interpreted as the character X. Outside a character class, these sequences have different meanings (see **Backslash and Nonprinting Characters** on page IV-168).

Backslash and Generic Character Types

The third use of backslash is for specifying generic character types. The following are always recognized:

Igor Pattern	PCRE Pattern	Character(s) Matched
\\d	\d	Any decimal digit
\\D	\D	Any character that is not a decimal digit
\\s	\s	Any whitespace character
\\S	\S	Any character that is not a whitespace character

Igor Pattern	PCRE Pattern	Character(s) Matched
\\w	\w	Any "word" character
\\W	\W	Any "nonword" character

Each pair of escape sequences, such as \d and \D, partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

The \s whitespace characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). In Igor Pro 6 \s did not match the VT character (code 11).

A "word" character is an underscore or any character that is a letter or digit.

Backslash and Simple Assertions

The fourth use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described in **Assertions** on page IV-179. The backslashed assertions are:

Igor Pattern	PCRE Pattern	Character(s) Matched
\\b	\b	At a word boundary
\\B	\B	Not at a word boundary
\\A	\A	At start of subject
\\Z	\Z	At end of subject or before newline at end
\\z	\z	At end of subject
\\G	\G	At first matching position in subject

These assertions may not appear in character classes (but note that \b has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match \w or \W (i.e. one matches \w and the other matches \W), or the start or end of the string if the first or last character matches \w, respectively.

While PCRE defines additional simple assertions (\A, \Z, \z and \G), they are not any more useful to Igor's regular expression commands than the ^ and $ characters.

Circumflex and Dollar

Outside a character class, in the default matching mode, the circumflex character ^ is an assertion that is true only if the current matching point is at the start of the subject string. Inside a character class, circumflex has an entirely different meaning (see **Character Classes and Brackets** on page IV-171).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an "anchored" pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character $ is an assertion that is true only if the current matching point is at the end of the subject string, or immediately before a newline character that is the last character in the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

Dot, Period, or Full Stop

Outside a character class, a dot in the pattern matches any one character in the subject, including a nonprinting character, but not (by default) newline. Dot has no special meaning in a character class.

The match option setting (?s) changes the default behavior of dot so that it matches any character including newline. The setting can appear anywhere before the dot in the pattern. See **Matching Newlines** on page IV-173 for details.

Character Classes and Brackets

An opening bracket introduces a character class which is terminated by a closing bracket. A closing bracket on its own is not special. If a closing bracket is required as a member of the class, it must be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject. A matched character must be in the set of characters defined by the class, unless the first character in the class definition is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class `[aeiou]` matches any English lower case vowel, whereas `[^aeiou]` matches any character that is not an English lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters that are in the class by enumerating those that are not.

When caseless matching is set, using the `(?i)` match option setting, any letters in a class represent both their upper case and lower case versions, so for example, the caseless pattern `(?i)[aeiou]` matches *A* as well as *a*, and the caseless pattern `(?i)[^aeiou]` does not match *A*.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, `[d-m]` matches any letter between *d* and *m*, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

To include a right-bracket in a range you must use `\]`. As usual, this would be represented in a literal Igor string as `\\]`.

Though it is rarely needed, you can specify a range using octal numbers, for example `[\000-\037]`.

The character types `\d`, `\D`, `\p`, `\P`, `\s`, `\S`, `\w`, and `\W` may also appear in a character class, and add the characters that they match to the class. For example, `[\dABCDEF]` matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class `[^\W_]` matches any letter or digit, but not underscore. The corresponding **Grep** command would begin with

```
Grep/E="[^\\W_]"…
```

The only metacharacters that are recognized in character classes are backslash, hyphen (only where it can be interpreted as specifying a range), circumflex (only at the start), opening bracket (only when it can be interpreted as introducing a POSIX class name — see **POSIX Character Classes** on page IV-171), and the terminating closing bracket. However, escaping other nonalphanumeric characters does no harm.

POSIX Character Classes

Perl supports the POSIX notation for character classes. This uses names enclosed by `[:` and `:]` within the enclosing brackets. PCRE also supports this notation. For example,

```
[01[:alpha:]%]
```

matches “0”, “1”, any alphabetic character, or “%”.

The supported class names, all of which must appear between [: and :] inside a character class specification, are

alnum	Letters and digits.
alpha	Letters.
ascii	Character codes 0 - 127.
blank	Space or tab only.
cntrl	Control characters.
digit	Decimal digits (same as \d).
graph	Printing characters, excluding space.
	[:graph:] matches all characters with the Unicode L, M, N, P, S, or Cf properties with a few arcane exceptions.
lower	Lower case letters.
print	Printing characters, including space.
	[:print:] matches the same characters as [:graph:] plus space characters that are not controls, that is, characters with the Unicode Zs property.
punct	Printing characters, excluding letters and digits.
	[:punct:] matches all characters that have the Unicode P (punctuation) property, plus those characters whose code points are less than 128 that have the S (Symbol) property.
space	White space (not quite the same as \s).
upper	Upper case letters.
word	"Word" characters (same as \w).
xdigit	Hexadecimal digits.

The "space" characters are horizontal tab (HT-9), linefeed (LF-10), vertical tab (VT-11), formfeed (FF-12), carriage-return (CR-13), and space (32).

The class name word is a Perl extension, and blank is a GNU extension from Perl 5.8. Another Perl extension is negation that is indicated by a ^ character after the colon. For example,

```
[12[:^digit:]]
```

matches "1", "2", or any nondigit. PCRE (and Perl) also recognize the POSIX syntax [.ch.] and [=ch=] where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

Alternation

Vertical bar characters are used to separate alternative patterns. For example, the pattern

```
gilbert|sullivan
```

matches either "gilbert" or "sullivan". Any number of alternative patterns may be specified, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined in **Subpatterns** on page IV-175), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

Match Option Settings

Character matching options can be changed from within the pattern by a sequence of Perl option letters enclosed between `(?` and `)`. The option letters are:

Option	PCRE Name	Characters Matched
i	PCRE_CASELESS	Upper and lower case.
m	PCRE_MULTILINE	^ matches start of string and just after a new line (\n). $ matches end of string and just before a new line. Without (?m), ^ and $ match only the start and end of the entire string.
s	PCRE_DOTALL	. matches all characters including newline. Without (?s), . does not match newlines. See **Matching Newlines** on page IV-173.
U	PCRE_UNGREEDY	Reverses the "greediness" of the quantifiers so that they are not greedy by default, but become greedy if followed by ?.

For example, `(?i)` sets caseless matching.

You can unset these options by preceding the letter with a hyphen: (?-i) turns off caseless matching.

You can combine a setting and unsetting such as `(?i-U)`, which sets PCRE_CASELESS while unsetting PCRE_UNGREEDY.

When an option change occurs at top level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows. If the change is placed right at the start of a pattern, PCRE extracts it into the global options.

An option change within a subpattern affects only that part of the current pattern that follows it, so

```
(a(?i)b)c
```

matches abc and aBc and no other strings.

Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

```
(a(?i)b|c)
```

matches “ab”, “aB”, “c”, and “C”, even though when matching “C” the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time.

The other PCRE 5-specific options, such as (?e), (?A), (?D), (?S), (?x) and (?X), are implemented but are not very useful with Igor's regular expression commands.

Matching Newlines

Igor generally uses carriage-return (ASCII code 13) return to indicate a new line in text. This is represented in a literal string by "\r". For example:

```
Print "Hello\rGoodbye"
```

prints two lines of text to the history area.

However, in regular expressions, the newline character is linefeed (ASCII code 10). This is represented in a literal string by "\n". Carriage-return has no special status in regular expressions.

The match option setting `(?s)` changes the default behavior of dot so that it matches any character including newline. The setting can appear anywhere before the dot in the pattern. For example:

```
Function DemoNewline(includeNewline)
   Variable includeNewline

   String subject = "Hello\nGoodbye"   // \n represents newline
```

```
   String regExp
   if (includeNewline)
      regExp = "(?s)(.+)"     // One or more of any character
   else
      regExp = "(.+)"         // One or more of any character except newline
   endif

   String result
   SplitString /E=(regExp) subject, result
   Print result
End
```

The output from DemoNewline(0) is:

```
   Hello
```

The output from DemoNewline(1) is:

```
   Hello\nGoodbye
```

Here is a more realistic example:

```
Function DemoDotAll()
   String theString = ExampleHTMLString()

   // This regular expression attempts to extract items from
   // HTML code for an ordered list. It fails because the HTML
   // code contains newlines and, by default, . does not match newlines.
   String regExpFails="<li>(.*?)</li>.*<li>(.*?)</li>"
   String str1, str2
   SplitString/E=regExpFails theString, str1, str2
   Print V_flag, " subpatterns were matched using regExpFails"
   Printf "\"%s\", \"%s\"\r", str1, str2

   // This regular expression works because the "(?s)" match
   // option setting causes . to match newlines.
   String regExpWorks="(?s)<li>(.*?)</li>.*<li>(.*?)</li>"
   SplitString/E=regExpWorks theString, str1, str2
   Print V_flag, " subpatterns were matched using regExpWorks"
   Printf "\"%s\", \"%s\"\r", str1, str2
End
Function/S ExampleHTMLString()   // Returns HTML code containing newlines
   String theString = ""
   theString += "<ol>\n"
   theString += "\t<li>item 1</li>\n"
   theString += "\t<li>item 2</li>\n"
   theString += "<\ol>\n"
   return theString
End
```

```
•DemoDotAll()
  0   subpatterns were matched using regExpFails
  "", ""
  2 subpatterns were matched using regExpWorks
  "item 1", "item 2"
```

Subpatterns

Subpatterns are used to group alternatives, to match a previously-matched pattern again, and to extract match text using **Demo**.

Subpatterns are delimited by parentheses, which can be nested. Turning part of a pattern into a subpattern localizes a set of alternatives. For example, this pattern (which includes two vertical bars signifying alternation):

```
cat(aract|erpillar|)
```

matches one of the words "cat", "cataract", or "caterpillar". Without the parentheses, it would match "cataract", "erpillar" or the empty string.

Named Subpatterns

Specifying subpatterns by number is simple, but it can be very hard to keep track of the numbers in complicated regular expressions. Furthermore, if an expression is modified, the numbers may change. To help with this difficulty, PCRE supports the naming of subpatterns, something that Perl does not provide. The Python syntax (`?P<`*name*`>`...) is used. For example:

```
My (?P<catdog>cat|dog) is cooler than your (?P=catdog)
```

Here catdog is the name of the first and only subpattern. `?P<catdog>` names the subpattern and `(?P=catdog)` matches the previous match for that subpattern.

Names consist of alphanumeric characters and underscores, and must be unique within a pattern.

Named capturing parentheses are still allocated numbers as well as names. This has the same effect as the previous example:

```
My (?P<catdog>cat|dog) is cooler than your \\1
```

Repetition

Repetition is specified by quantifiers:

?	0 or 1 quantifier
	Example: `[abc]?` - Matches 0 or 1 occurrences of a or b or c
*	0 or more quantifier
	Example: `[abc]*` - Matches 0 or more occurrences of a or b or c
+	1 or more quantifier
	Example: `[abc]+` - Matches 1 or more occurrences of a or b or c
{n}	n times quantifier
	Example: `[abc]{3}` - Matches 3 occurrences of a or b or c
{n,m}	n to m times quantifier
	Example: `[abc]{3,5}` - Matches 3 to 5 occurrences of a or b or c

Quantifiers can follow any of the following items:

- A literal data character
- The . metacharacter
- The \C escape sequence
- An escape such as \d that matches a single character
- A character class

- A back reference (see **Back References** on page IV-178)
- A parenthesized subpattern (unless it is an assertion)

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in braces, separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

```
z{2,4}
```

matches "zz", "zzz", or "zzzz".

If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

```
[aeiou]{3,}
```

matches at least 3 successive vowels, but may match many more, while

```
\d{8}
```

matches exactly 8 digits. A left brace that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, `{,6}` is not a quantifier, but a literal string of four characters.

Quantifiers apply to characters rather than to individual data units. Thus, for example, `\x{100}{2}` matches two characters, each of which is represented by a two-byte sequence in a UTF-8 string. Similarly, `\X{3}` matches three Unicode extended grapheme clusters, each of which may be several data units long (and they may be of different lengths).

The quantifier `{0}` is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience (and historical compatibility) the three most common quantifiers have single-character abbreviations:

`*`	Equivalent to `{0,}`
`+`	Equivalent to `{1,}`
`?`	Equivalent to `{0,1}`

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

```
(a?)*
```

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

Quantifier Greediness

By default, the quantifiers are "greedy", that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between `/*` and `*/` and within the comment, individual `*` and `/` characters may appear. An attempt to match C comments by applying the pattern

```
/\*.*\*/        or    Grep/E="/\\*.*\\*/"
```

to the string

```
/* first comment */  not comment  /* second comment */
```

fails because it matches the entire string owing to the greediness of the `.*` item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

```
/\*.*?\*/       or    Grep/E="/\\*.*?\\*/"
```

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches.

Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

```
\d??\d          or    Grep/E="\\d??\\d"
```

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the PCRE_UNGREEDY option `(?U)` is set, the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behavior.

Quantifiers With Subpatterns

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more memory is required for the compiled pattern, in proportion to the size of the minimum or maximum.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after

```
(tweedle[dume]{3}\s*)+   or    Grep/E="(tweedle[dume]{3}\\s*)+"
```

has matched “tweedledum tweedledee” the value of the captured substring is “tweedledee”. However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

```
/(a|(b))+/
```

matches “aba” the value of the second captured substring is “b”.

Atomic Grouping and Possessive Quantifiers

With both maximizing and minimizing repetition, failure of what follows normally reevaluates the repeated item to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern `\d+foo` when applied to the subject line

```
123456bar
```

After matching all 6 digits and then failing to match “foo”, the normal action of the matcher is to try again with only 5 digits matching the `\d+` item, and then with 4, and so on, before ultimately failing. “Atomic grouping” provides the means for specifying that once a subpattern has matched, it is not to be reevaluated in this way.

If we use atomic grouping for the previous example, the matcher would give up immediately on failing to match “foo” the first time. The notation is a kind of special parenthesis, starting with `(?>` as in this example:

```
(?>\d+)foo      or    Grep/E="(?>\\d+)foo"
```

This kind of parenthesis “locks up” the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Atomic grouping subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both `\d+` and `\d+?`

are prepared to adjust the number of digits they match in order to make the rest of the pattern match, `(?>\d+)` can only match an entire sequence of digits.

Atomic groups in general can of course contain arbitrarily complicated subpatterns, and can be nested. However, when the subpattern for an atomic group is just a single repeated item, as in the example above, a simpler notation, called a "possessive quantifier" can be used. This consists of an additional + character following a quantifier. Using this notation, the previous example can be rewritten as

```
\d++foo          or    Grep/E="\\d++foo"
```

Possessive quantifiers are always greedy; the setting of the PCRE_UNGREEDY option is ignored. They are a convenient notation for the simpler forms of atomic group. However, there is no difference in the meaning or processing of a possessive quantifier and the equivalent atomic group.

The possessive quantifier syntax is an extension to the Perl syntax. It originates in Sun's Java package.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of an atomic group is the only way to avoid some failing matches taking a very long time indeed. The pattern

```
(\D+|<\d+>)*[!?]   or    Grep/E="(\\D+|<\\d+>)*[!?]"
```

matches an unlimited number of substrings that either consist of nondigits, or digits enclosed in <>, followed by either ! or ?. When it matches, it runs quickly. However, if it is applied to

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the internal `\D+` repeat and the external * repeat in a large number of ways, and all have to be tried. (The example uses `[!?]` rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed so that it uses an atomic group, like this:

```
((?>\D+)|<\d+>)*[!?]     or    Grep/E="((?>\\D+)|<\\d+>)*[!?]"
```

sequences of nondigits cannot be broken, and failure happens quickly.

Back References

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (that is, to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. See **Backslash and Nonprinting Characters** on page IV-168 for further details of the handling of digits following a backslash.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself (see **Subpatterns as Subroutines** on page IV-183 for a way of doing that). So the pattern

```
(sens|respons)e and \1ibility or /E="(sens|respons)e and \\1ibility"
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If caseful matching is in force at the time of the back reference, the case of letters is relevant. For example,

```
((?i)rah)\s+\1     or    Grep/E="((?i)rah)\\s+\\1"
```

matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

Back references to named subpatterns use the Python syntax `(?P<name>)`. We could rewrite the above example as follows:

```
(?P<p1>(?i)rah)\s+(?P=p1)   or    Grep/E="(?P<p1>(?i)rah)\\s+(?P=p1)"
```

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail. For example, the pattern

```
(a|(bc))\2
```

always fails if it starts to match "a" rather than "bc". Because there may be many capturing parentheses in a pattern, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. An empty comment (see **Regular Expression Comments** on page IV-182) can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, `(a\1)` never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern

```
(a|b\1)+        or    Grep/E="(a|b\\1)+"
```

matches any number of *a*'s and also "aba", "ababbaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

Assertions

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as `\b`, `\B`, `^` and `$` are described in **Backslash and Simple Assertions** on page IV-170.

More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it. An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed.

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

Lookahead Assertions

Lookahead assertions start with `(?=` for positive assertions and `(?!` for negative assertions. For example,

```
\w+(?=;)        or    Grep/E="\\w+(?=;)"
```

matches a word followed by a semicolon, but does not include the semicolon in the match, and

```
foo(?!bar)
```

matches any occurrence of "foo" that is not followed by "bar". Note that the apparently similar pattern

```
(?!foo)bar
```

does not find an occurrence of "bar" that is preceded by something other than "foo"; it finds any occurrence of "bar" whatsoever, because the assertion `(?!foo)` is always true when the next three characters are "bar". A lookbehind assertion is needed to achieve the other effect.

If you want to force a matching failure at some point in a pattern, the most convenient way to do it is with `(?!)` because an empty string always matches, so an assertion that requires there not to be an empty string must always fail.

Lookbehind Assertions

Lookbehind assertions start with `(?<=` for positive assertions and `(?<!` for negative assertions. For example,

```
(?<!foo)bar
```

does find an occurrence of "bar" that is not preceded by "foo". The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several alternatives, they do not all have to have the same fixed length. Thus

```
(?<=bullock|donkey)
```

is permitted, but

```
(?<!dogs?|cats?)
```

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl (at least for 5.8), which requires all branches to match the same length of string. An assertion such as

```
(?<=ab(c|de))
```

is not permitted, because its single top-level branch can match two different lengths, but it is acceptable if rewritten to use two top-level branches:

```
(?<=abc|abde)
```

In some cases, the escape sequence \K (see above) can be used instead of a lookbehind assertion to get round the fixed-length restriction.

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed width and then try to match. If there are insufficient characters before the current position, the match is deemed to fail.

PCRE does not allow the \C escape (which matches a single data unit) to appear in lookbehind assertions, because it makes it impossible to calculate the length of the lookbehind. The \X and \R escapes, which can match different numbers of data units, are also not permitted.

Atomic groups can be used in conjunction with lookbehind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as

```
abcd$
```

when applied to a long string that does not match. Because matching proceeds from left to right, PCRE will look for each *a* in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as

```
^.*abcd$
```

the initial `.*` matches the entire string at first, but when this fails (because there is no following *a*), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for *a* covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

```
^(?>.*)(?<=abcd)
```

or, equivalently, using the possessive quantifier syntax,

```
^.*+(?<=abcd)
```

there can be no backtracking for the `.*` item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

Using Multiple Assertions

Several assertions (of any sort) may occur in succession. For example,

```
(?<=\d{3})(?<!999)foo    or    Grep/E="(?<=\\d{3})(?<!999)foo"
```

matches “foo” preceded by three digits that are not “999”. Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not “999”. This pattern does not match “foo” preceded by six characters, the first of which are digits and the last three of which are not “999”. For example, it doesn’t match “123abcfoo”. A pattern to do that is

```
(?<=\d{3}...)(?<!999)foo or    Grep/E="(?<=\\d{3}...)(?<!999)foo"
```

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not “999”.

Assertions can be nested in any combination. For example,

```
(?<=(?<!foo)bar)baz
```

matches an occurrence of “baz” that is preceded by “bar” which in turn is not preceded by “foo”, while

```
(?<=\d{3}(?!999)...)foo or Grep/E=" (?<=\\d{3}(?!999)...)foo"
```

is another pattern that matches “foo” preceded by three digits and any three characters that are not “999”.

Conditional Subpatterns

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a previous capturing subpattern matched or not. The two possible forms of conditional subpattern are

```
(?(condition)yes-pattern)
(?(condition)yes-pattern|no-pattern)
```

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are three kinds of condition. If the text between the parentheses consists of a sequence of digits, the condition is satisfied if the capturing subpattern of that number has previously matched. The number must be greater than zero. Consider the following pattern, which contains nonsignificant white space to make it more readable and to divide it into three parts for ease of discussion:

```
( \( )?    [^()]+    (?(1) \) )
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of nonparentheses, optionally enclosed in parentheses.

If the condition is the string `(R)`, it is satisfied if a recursive call to the pattern or subpattern has been made. At “top level”, the condition is false. This is a PCRE extension. Recursive patterns are described in the next section.

If the condition is not a sequence of digits or `(R)`, it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing nonsignificant white space, and with the two alternatives on the second line:

```
(?(?=[^a-z]*[a-z])
\d{2}-[a-z]{3}-\d{2}  |  \d{2}-\d{2}-\d{2})
```

The condition is a positive lookahead assertion that matches an optional sequence of nonletters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms `dd-aaa-dd` or `dd-dd-dd`, where `aaa` are letters and `dd` are digits.

Regular Expression Comments

The sequence (?# marks the start of a comment that continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

Recursive Patterns

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth. Perl provides a facility that allows regular expressions to recurse (amongst other things). It does this by interpolating Perl code in the expression at runtime, and the code can refer to the expression itself. A Perl pattern to solve the parentheses problem can be created like this:

```
$re = qr{\( (?: (?>[^()]+) | (?p{$re}) )* \)}x;
```

The (?p{...}) item interpolates Perl code at runtime, and in this case refers recursively to the pattern in which it appears. Obviously, PCRE cannot support the interpolation of Perl code. Instead, it supports some special syntax for recursion of the entire pattern, and also for individual subpattern recursion.

The special item that consists of (? followed by a number greater than zero and a closing parenthesis is a recursive call of the subpattern of the given number, provided that it occurs inside that subpattern. (If not, it is a "subroutine" call, which is described in **Subpatterns as Subroutines** on page IV-183.) The special item (?R) is a recursive call of the entire regular expression.

For example, this PCRE pattern solves the nested parentheses problem (additional nonfunction whitespace has been added to separate the expression into parts):

```
\( ( (?>[^()]+) | (?R) )* \)
```

First it matches an opening parenthesis. Then it matches any number of substrings which can either be a sequence of nonparentheses, or a recursive match of the pattern itself (that is a correctly parenthesized substring). Finally there is a closing parenthesis.

If this were part of a larger pattern, you would not want to recurse the entire pattern, so instead you could use this:

```
( \( ( (?>[^()]+) | (?1) )* \))
```

We have put the pattern into parentheses, and caused the recursion to refer to them instead of the whole pattern. In a larger pattern, keeping track of parenthesis numbers can be tricky. It may be more convenient to use named parentheses instead. For this, PCRE uses (?P>name), which is an extension to the Python syntax that PCRE uses for named parentheses (Perl does not provide named parentheses). We could rewrite the above example as follows:

```
(?P<pn> \( ( (?>[^()]+) | (?P>pn) )* \))
```

This particular example pattern contains nested unlimited repeats, and so the use of atomic grouping for matching strings of nonparentheses is important when applying the pattern to strings that do not match. For example, when this pattern is applied to

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa()
```

it yields "no match" quickly. However, if atomic grouping is not used, the match runs for a very long time indeed because there are so many different ways the + and * repeats can carve up the subject, and all have to be tested before failure can be reported.

At the end of a match, the values set for any capturing subpatterns are those from the outermost level of the recursion at which the subpattern value is set. If you want to obtain intermediate values, a callout function can be used (see **Subpatterns as Subroutines** on page IV-183). If the pattern above is matched against

```
(ab(cd)ef)
```

the value for the capturing parentheses is "ef", which is the last value taken on at the top level. If additional parentheses are added, giving

```
\(( ( (?>[^()]+) | (?R) )* ) \)
     ↑                     ↑
```

the string they capture is "ab(cd)ef", the contents of the top level parentheses. If there are more than 15 capturing parentheses in a pattern, PCRE has to obtain extra memory to store data during a recursion, which it does by using pcre_malloc, freeing it via pcre_free afterward. If no memory can be obtained, the match fails with the PCRE_ERROR_NOMEMORY error.

Do not confuse the `(?R)` item with the condition `(R)`, which tests for recursion. Consider this pattern, which matches text in angle brackets, allowing for arbitrary nesting. Only digits are allowed in nested brackets (that is, when recursing), whereas any characters are permitted at the outer level.

```
< (?: (?(R) \d++  | [^<>]*+) | (?R)) * >
```

In this pattern, `(?(R)` is the start of a conditional subpattern, with two different alternatives for the recursive and nonrecursive cases. The `(?R)` item is the actual recursive call.

Subpatterns as Subroutines

If the syntax for a recursive subpattern reference (either by number or by name) is used outside the parentheses to which it refers, it operates like a subroutine in a programming language. An earlier example pointed out that the pattern

```
(sens|respons)e and \1ibility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If instead the pattern

```
(sens|respons)e and (?1)ibility
```

is used, it does match "sense and responsibility" as well as the other two strings. Such references must, however, follow the subpattern to which they refer.

Regular Expressions References

The regular expression syntax supported by **Grep**, **GrepString**, **GrepList**, and **Demo** is based on the *PCRE — Perl-Compatible Regular Expression Library* by Philip Hazel, University of Cambridge, Cambridge, England. The PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5.

Visit <http://pcre.org/> for more information about the PCRE library, and <http://www.perldoc.com/> for more about Perl regular expressions. This description, Regular Expressions In Igor, is taken from the PCRE documentation.

A good introductory book on regular expressions is: Forta, Ben, *Regular Expressions in 10 Minutes*, Sams Publishing, 2004.

A good comprehensive book on regular expressions is: Friedl, Jeffrey E. F., *Mastering Regular Expressions*, 2nd ed., 492 pp., O'Reilly Media, 2002.

A good web site is: http://www.regular-expressions.info

Working with Files

Here are the built-in operations that you can use to read from or write to files:

Operation	What It Does
Open	Opens an existing file for reading or writing. Can also create a new file. Can also append to an existing file. Returns a file reference number that you must pass to the other file operations. Use Open/D to present a dialog that allows the user to choose a file without actually opening the file.

Operation	What It Does
fprintf	Writes formatted text to an open file.
wfprintf	Writes wave data as formatted text to an open file.
FGetPos	Gets the position at which the next file read or write will be done.
FSetPos	Sets the position at which the next file read or write will be done.
FStatus	Given a file reference number, returns miscellaneous information about the file.
FBinRead	Reads binary data from a file into an Igor string, variable or wave. This is used mostly to read nonstandard file formats.
FBinWrite	Writes binary data from an Igor string, variable or wave. This is used mostly to write nonstandard file formats.
FReadLine	Reads a line of text from a text file into an Igor string variable. This can be used to parse an arbitrary format text file.
Close	Closes a file opened by the Open operation.

Before working with a file, you must use the Open operation to obtain a file reference number that you use with all the remaining commands. The Open operation creates new files, appends data to an existing file, or reads data from an existing file. There is no facility to deal with the resource fork of a Macintosh file.

Sometimes you may write a procedure that uses the Open operation and the Close operation but, because of an error, the Close operation never gets to execute. You then correct the procedure and want to rerun it. The file will still be open because the Close operation never got a chance to run. In this case, execute:

```
Close/A
```

from the command line to close all open files.

Finding Files

Two functions are provided to help you to determine what files exist in a particular folder:

- **TextFile**
- **IndexedFile**

IndexedFile is a more general version of TextFile.

You can also use the Open operation with the /D flag to present an open dialog.

Other File– and Folder–Related Operations and Functions

Igor Pro also supports a number of operations and functions for file or folder manipulation:

CopyFile	CopyFolder
DeleteFile	DeleteFolder
GetFileFolderInfo	SetFileFolderInfo
MoveFile	MoveFolder
CreateAliasShortcut	
NewPath	PathInfo
ParseFilePath	

Warning: Use caution when writing code that deletes or moves files or folders. These actions are not undoable.

Because the DeleteFolder, CopyFolder and MoveFolder operations have the potential to do a lot of damage if used incorrectly, they require user permission before overwriting or deleting a folder. The user controls the permission process using the Miscellaneous Settings dialog (Misc menu).

Writing to a Text File

You can generate output text files from Igor procedures or from the command line in formats acceptable to other programs. To do this, you need to use the **Open** and **Close** operations and the **fprintf** operation (page V-223) and the **wfprintf** operation (page V-942).

The following commands illustrate how you could use these operations to create an output text file. Each operation is described in detail in following sections:

```
Function WriteFileDemo(pathName, fileName)
   String pathName   // Name of Igor symbolic path (see Symbolic Paths)
   String fileName   // File name, partial path or full path

   Variable refNum
   Open refNum as "A New File"                // Create and open file
   fprintf refNum, "wave1\twave2\twave3\r"   // Write column headers
   Wave wave1, wave2, wave3
   wfprintf refNum, "" wave1, wave2, wave3   // Write wave data
   Close refNum                               // Close file
End
```

Open and Close Operations

You use the **Open** operation (page V-604) to open a file. For our purposes, the syntax of the Open operation is:

```
Open [/R/A/P=pathName/M=messageStr] variableName [as "filename"]
```

variableName is the name of a numeric variable. The Open operation puts a file reference number in that variable. You use the reference number to access the file after you've opened it.

A file specifications consists of a path (directions for finding a folder) and a file name. In the Open operation, you can specify the path in three ways:

Using a full path as the filename parameter:

```
Open refNum as "hd:Data:Run123.dat"
```

Or using a symbolic path name and a file name:

```
Open/P=DataPath refNum as "Run123.dat"
```

Or using a symbolic path name and a partial path including the file name:

```
Open/P=HDPath refNum as ":Data:Run123.dat"
```

A symbolic path is a short name that refers to a folder on disk. See **Symbolic Paths** on page II-21.

If you do not provide enough information to find the folder and file of interest, Igor displays a dialog which lets you select the file to open. If you supply sufficient information, Igor open the file without displaying the dialog.

To open an existing file for reading, use the /R flag. To append new data to an existing file, use the /A flag. If you omit both of these flags and if the file already exists, you overwrite any data that is in the file.

If you open a file for writing (you don't use /R) then, if there exists a file with the specified name, Igor opens the file and overwrites the existing data in the file. If there is no file with the specified name, Igor creates a new file.

Warning: If you're not careful you can inadvertently lose data using the Open operation by opening for writing without using the /A flag. To avoid this, use the /R (open for read) or /A (append) flags.

Wave Reference Functions

It is common to write a user-defined function that operates on all of the waves in a data folder, on the waves displayed in a graph or table, or on a wave identified by a cursor in a graph. For these purposes, you need to use wave reference functions.

Wave reference functions are built-in Igor functions that return a reference that can be used in a user-defined function. Here is an example that works on the top graph. Cursors are assumed to be placed on a region of a trace in the graph.

```
Function WaveAverageBetweenCursors()
   WAVE/Z w = CsrWaveRef(A)
   if (!WaveExists(w))        // Cursor is not on any wave.
      return NaN
   endif

   Variable xA = xcsr(A)
   Variable xB = xcsr(B)
   Variable avg = mean(w, xA, xB)

   return avg
End
```

CsrWaveRef returns a wave reference that identifies the wave a cursor is on.

An older function, CsrWave, returns the *name* of the wave the cursor is on. It would be tempting to use this to determine the wave the cursor is on, but it would be incorrect. The name of a wave by itself does not uniquely identify a wave because it does not specify the data folder in which the wave resides. For this reason, we usually need to use the wave reference function CsrWaveRef instead of CsrWave.

This example uses a wave reference function to operate on the waves displayed in a graph:

```
Function SmoothWavesInGraph()
   String list = TraceNameList("", ";", 1)
   String traceName
   Variable index = 0
   do
      traceName = StringFromList(index, list)
      if (strlen(traceName) == 0)
         break          // No more traces.
      endif
      WAVE w = TraceNameToWaveRef("", traceName)
      Smooth 5, w
      index += 1
   while(1)
End
```

Use **WaveRefIndexedDFR** to iterate over the waves in a given data folder.

Here are the wave reference functions. See Chapter V-1, **Igor Reference**, for details on each of them.

Function	Comment
CsrWaveRef	Returns a reference to the Y wave to which a cursor is attached.
CsrXWaveRef	Returns a reference to the X wave when a cursor is attached to an XY pair.
WaveRefIndexedDFR	Returns a reference to a wave in the specified data folder.
WaveRefIndexed	Returns a reference to a wave in a graph or table or to a wave in the current data folder.
XWaveRefFromTrace	Returns a reference to an X wave in a graph. Used with the output of TraceNameList.

Function	Comment
ContourNameToWaveRef	Returns a reference to a wave displayed as a contour plot. Used with the output of ContourNameList.
ImageNameToWaveRef	Returns a reference to a wave displayed as an image. Used with the output of ImageNameList.
TraceNameToWaveRef	Returns a reference to a wave displayed as a waveform or as the Y wave of an XY pair in a graph. Used with the output of TraceNameList.
TagWaveRef	Returns a reference to a wave to which a tag is attached in a graph. Used in creating a smart tag.
WaveRefsEqual	Returns the truth two wave references are the same.
WaveRefWaveToList	Returns a semicolon-separated string list containing the full or partial path to the wave referenced by each element of the input wave reference wave.
ListToWaveRefWave	Returns a free wave containing a wave reference for each of the waves in the semicolon-separated input string list.

This simple example illustrates the use of the CsrWaveRef wave reference function:

```
Function/WAVE CursorAWave()
   WAVE/Z w = CsrWaveRef(A)
   return w
End

Function DemoCursorAWave()
   WAVE/Z w = $CursorAWave()
   if (WaveExists(w)==0)
      Print "oops: no wave"
   else
      Printf "Cursor A is on the wave '%s'\r", NameOfWave(w)
   endif
End
```

Processing Lists of Waves

Igor users often want to use a string list of waves in places where Igor is looking for just the name of a single wave. For example, they would like to do this:

```
Display "wave0;wave1;wave2"
```

or, more generally:

```
Function DisplayListOfWaves(list)
   String list                // e.g., "wave0;wave1;wave2"

   Display $list
End
```

Unfortunately, Igor can't handle this. However, there are techniques for achieving the same result.

Graphing a List of Waves

This example illustrates the basic technique for processing a list of waves.

```
Function DisplayWaveList(list)
   String list             // A semicolon-separated list.

   Variable index = 0
   do
      // Get the next wave name
      String name = StringFromList(index, list)
      if (strlen(name) == 0)
```

```
         break                        // Ran out of waves
      endif

      Wave w = $name
      if (index == 0)                 // Is this the first wave?
         Display w
      else
         AppendToGraph w
      endif
      index += 1
   while (1)          // Loop until break above
End
```

To make a graph of all of the waves in the current data folder, you could execute

```
DisplayWaveList(WaveList("*", ";", ""))
```

Operating on the Traces in a Graph

In a previous section, we showed an example that operates on the waves displayed in a graph. It used a wave reference function, TraceNameToWaveRef. If you want to write a function that operates on *traces* in a graph, you would *not* use wave reference functions. That's because Igor operations that operate on *traces* expect *trace names*, not wave references. For example:

```
Function GrayOutTracesInGraph()
   String list = TraceNameList("", ";", 1)
   Variable index = 0
   do
      String traceName = StringFromList(index, list)
      if (strlen(traceName) == 0)
         break                        // No more traces.
      endif

      // WRONG: ModifyGraph expects a trace name and w is not a trace name
      WAVE w = TraceNameToWaveRef("", traceName)
      ModifyGraph rgb(w)=(50000,50000,50000)

      // RIGHT
      ModifyGraph rgb($traceName)=(50000,50000,50000)

      index += 1
   while(1)
End
```

Using a Fixed-Length List

In the previous examples, the number of waves in the list was unimportant and all of the waves in the list served the same purpose. In this example, the list has a fixed number of waves and each wave has a different purpose.

```
Function DoLineFit(list)
   String list          // List of waves names: source, weight

   // Pick out the expected wave names

   String sourceStr = StringFromList(0, list)
   Wave source = $sourceStr

   String weightStr = StringFromList(2, list)
   Wave weight = $weightStr

   CurveFit line source /D /W=weight
End
```

You could invoke this function as follows:

```
DoLineFit("wave0;wave1;")
```

For most purposes, it is better to design the function to take wave reference parameters rather than a string list.

Operating on Qualified Waves

This example illustrates how to operate on waves that match a certain criterion. It is broken into two functions - one that creates the list of qualified waves and a second that operates on them. This organization gives us a general purpose routine (ListOfMatrices) that we would not have if we wrote the whole thing as one function.

```
Function/S ListOfMatrices()
   String list = ""
   Variable index=0
   do
      WAVE/Z w=WaveRefIndexedDFR(:,index)    // Get next wave.
      if (WaveExists(w) == 0)
         break                               // No more waves.
      endif
      if (WaveDims(w) == 2)
         // Found matrix. Add to list with separator.
         list += NameOfWave(w) + ";"
      endif
      index += 1
   while(1)             // Loop till break above.
   return list
End

Function ChooseAndDisplayMatrix()
   String theList = ListOfMatrices()

   String theMatrix
   Prompt theMatrix, "Matrix to display:", popup theList
   DoPrompt "Display Matrix", theMatrix
   if (V_Flag != 0)
      return -1
   endif

   WAVE m = $theMatrix
   NewImage m
End
```

In the preceding example, we needed a list of wave names in a string to use in a Prompt statement. More often we want a list of wave references on which to operate. The next example illustrates how to do this using a general purpose routine that returns a list of wave references in a free wave:

```
// Returns a free wave containing wave references
// for each 2D wave in the current data folder
Function/WAVE GetMatrixWavesInCDF()
   Variable numWavesInCDF = CountObjects(":", 1)
   Make/FREE/WAVE/N=(numWavesInCDF) list

   Variable numMatrixWaves = 0
   Variable i
   for(i=0; i<numWavesInCDF; i+=1)
      WAVE w = WaveRefIndexedDFR(:,i)
      Variable numDimensions = WaveDims(w)
      if (numDimensions == 2)
         list[numMatrixWaves] = w
         numMatrixWaves += 1
      endif
```

```
   endfor

   Redimension/N=(numMatrixWaves) list
   return list       // Ownership of the wave is passed to the calling routine
End

Function DemoMatrixWaveList()
   Wave/WAVE freeList = GetMatrixWavesInCDF()   // We now own the freeList wave
   Variable numMatrixWaves = numpnts(freeList)
   Printf "Number of matrix waves in current data folder: %d\r", numMatrixWaves

   Variable i
   for(i=0; i<numMatrixWaves; i+=1)
      Wave w = freeList[i]
      String name = NameOfWave(w)
      name = PossiblyQuoteName(name)
      Printf "Wave %d: %s\r", i, name
   endfor

   // freeList is automatically killed when the function returns
End
```

The ExecuteCmdOnList Function

The ExecuteCmdOnList function is implemented by a WaveMetrics procedure file and executes any command for each wave in the list. For example, the following commands do a WaveStats operation on each wave.

```
Make wave0=gnoise(1), wave1=gnoise(10), wave2=gnoise(100)
ExecuteCmdOnList("WaveStats %s", "wave0;wave1;wave2;")
```

The ExecuteCmdOnList function is supplied in the "Execute Cmd On List" procedure file. See **The Include Statement** on page IV-155 for instructions on including a procedure file.

This technique is on the kludgy side. If you can achieve your goal in a more straightforward fashion without heroic efforts, you should use regular programming techniques.

The Execute Operation

Execute is a built-in operation that executes a string expression as if you had typed it into the command line. The main purpose of Execute is to get around the restrictions on calling macros and external operations from user functions.

We try to avoid using Execute because it makes code obscure and difficult to debug. If you can write a procedure without Execute, do it. However, there are some cases where using Execute can save pages of code or achieve something that would otherwise be impossible. For example, the TileWindow operation can not be directly called from a user-defined function and therefore must be called using Execute.

It is a good idea to compose the command to be executed in a local string variable and then pass that string to the Execute operation. Use this to print the string to the history for debugging. For example:

```
Function DemoExecute(list)
   String list    // Semicolon-separated list of names of windows to tile
   list = ReplaceString(";", list, ",")   // We need commas for TileWindows
   list = RemoveEnding(list, ",")         // Remove trailing comma, if any
   String cmd
   sprintf cmd, "TileWindows %s", list
   Print cmd                              // For debugging only
   Execute cmd
End
```

When you use Execute, you must be especially careful in the handling of wave names. See **Programming with Liberal Names** on page IV-157 for details.

When Execute runs, it is as if you typed a command in the command line. Local variables in macros and functions are not accessible. The example in **Calling an External Operation From a User-Defined Function** on page IV-191 shows how to use the sprintf operation to solve this problem.

Using a Macro From a User-Defined Function

A macro can not be called directly from a user function. To do so, we must use Execute. This is a trivial example for which we would normally not resort to Execute but which clearly illustrates the technique.

```
Function Example()
   Make wave0=enoise(1)
   Variable/G V_avg              // Create a global
   Execute "MyMacro(\"wave0\")"  // Invokes MyMacro("wave0")
   return V_avg
End

Macro MyMacro(wv)
   String wv
   WaveStats $wv     // Sets global V_avg and 9 other local vars
End
```

Execute does not supply good error messages. If the macro generates an error, you may get a cryptic message. Therefore, debug the macro *before* you call it with the Execute operation.

Calling an External Operation From a User-Defined Function

Prior to Igor Pro 5, external operations could not be called directly from user-defined functions and had to be called via Execute. Now it is possible to write an external operation so that it can be called directly. However, very old XOPs that have not been updated still need to be called through Execute. This example shows how to do it.

If you attempt to directly use an external operation which does not support it, Igor displays an error dialog telling you to use Execute for that operation.

The external operation in this case is VDTWrite which sends text to the serial port. It is implemented by the VDT XOP (no longer shipped as of Igor7).

```
Function SetupVoltmeter(range)
   Variable range    // .1, .2, .5, 1, 2, 5 or 10 volts

   String voltmeterCmd
   sprintf voltmeterCmd, "DVM volts=%g", range
   String vdtCmd
   sprintf vdtCmd "VDTWrite \"%s\"\r\n", voltmeterCmd
   Execute vdtCmd
End
```

In this case, we are sending the command to a voltmeter that expects something like:

```
DVM volts=.2<CR><LF>
```

to set the voltmeter to the 0.2 volt range.

The parameter that we send to the Execute operation is:

```
VDTWrite "DVM volts=.2\r\n"
```

The backslashes used in the second sprintf call insert two quotation marks, a carriage return, and a linefeed in the command about to be executed.

A newer VDT2 XOP exists which includes external operations that *can* be directly called from user-functions. Thus, new programming should use the VDT2 XOP and will not need to use Execute.

Other Uses of Execute

Execute can also accept as an argument a string variable containing commands that are algorithmically constructed. Here is a simple example:

```
Function Fit(w, fitType)
   WAVE w               // Source wave
   String fitType       // One of the built-in Igor fit types

   String name = NameOfWave(w)
   name = PossiblyQuoteName(name)   // Liberal names need quotes

   String cmd
   sprintf cmd, "CurveFit %s %s", fitType, name
   Execute cmd
End
```

Use this function to do any of the built-in fits on the specified wave. Without using the Execute operation, we would have to write it as follows:

```
Function Fit(w, fitType)
   WAVE w                  // Source wave
   String fitType          // One of the built-in Igor fit types

   strswitch(fitType)
      case "line":
         CurveFit line w
         break

      case "exp":
         CurveFit exp w
         break

      <and so on for each fit type>
   endswitch
End
```

Note the use of sprintf to prepare the string containing the command to be executed. The following would not work because when Execute runs, local variables and parameters are not accessible:

```
Execute "CurveFit fitType name"
```

Deferred Execution Using the Operation Queue

It is sometimes necessary to execute a command after the current function has finished. This is done using the Execute/P operation. For details, see **Operation Queue** on page IV-263.

Procedures and Preferences

As explained under **Preferences** on page III-453, Igor allows you to set many preferences. Most of the preferences control the style of new objects, including new graphs, traces and axes added to graphs, new tables, columns added to tables, and so on.

Preferences are usually used only for manual "point-and-click" operation. We usually don't want preferences to affect the behavior of procedures. The reason for this is that we want a procedure to do the same thing no matter who runs it. Also, we want it to do the same thing tomorrow as it does today. If we allowed preferences to take effect during procedure execution, a change in preferences could change the effect of a procedure, making it unpredictable.

By default, preferences do not take effect during procedure execution. If you want to override the default behavior, you can use the Preferences operation. From within a procedure, you can execute `Preferences 1` to turn preferences on. This affects the procedure and any subroutines that it calls. It stays in effect until you execute `Preferences 0`.

When a *macro* ends, the state of preferences reverts to what it was when that macro started. If you change the preference setting within a *function*, the preferences state does *not* revert when that function ends. You must turn preferences on, save the old preferences state, execute Igor operations, and then restore the preferences state. For example:

```
Function DisplayWithPreferences(w)
   Wave w

   Variable oldPrefState
   Preferences 1                    // Turn preferences on
   oldPrefState = V_Flag            // Save the old state

   Display w                        // Create graph using preferences

   Preferences oldPrefState         // Restore old prefs state
End
```

Experiment Initialization Procedures

When Igor loads an experiment, it checks to see if there are any commands in the procedure window before the first macro, function or menu declaration. If there are such commands Igor executes them. This provides a way for you to initialize things. These initialization commands can invoke procedures that are declared later in the procedure window.

Also see **BeforeFileOpenHook** on page IV-271 and **IgorStartOrNewHook** on page IV-275 for other initialization methods.

Procedure Subtypes

A procedure subtype identifies the purpose for which a particular procedure is intended and the appropriate menu from which it can be chosen. For example, the Graph subtype puts a procedure in the Graph Macros submenu of the Windows menu.

```
Window Graph0() : Graph
   PauseUpdate; Silent 1      // building window...
   Display/W=(5,42,400,250) wave0,wave1,wave2
   <more commands>
End
```

When Igor automatically creates a procedure, for example when you close and save a graph, it uses the appropriate subtype. When you create a curve fitting function using the Curve Fitting dialog, the dialog automatically uses the FitFunc subtype. You usually don't need to use subtypes for procedures that you write.

This table shows the available subtypes and how they are used.

Subtype	Effect	Available for
`Graph`	Displayed in Graph Macros submenu.	Macros
`GraphStyle`	Displayed in Graph Macros submenu and in Style pop-up menu in New Graph dialog.	Macros
`GraphMarquee`	Displayed in graph marquee. This keyword is no longer recommended. See **Marquee Menu as Input Device** on page IV-151 for details.	Macros and functions
`CursorStyle`	Displayed in Style Function submenu of cursor pop-up menu in graph info pane.	Functions
`DrawUserShape`	Marks a function as suitable for drawing a user-defined drawing object. See **DrawUserShape**.	Functions
`GridStyle`	Displayed in Grid-Style submenu of mover pop-up menu in drawing tool palette.	Functions

Subtype	Effect	Available for
Table	Displayed in Table Macros submenu.	Macros
TableStyle	Displayed in Table Macros submenu and in Style pop-up menu in New Table dialog.	Macros
Layout	Displayed in Layout Macros submenu.	Macros
LayoutStyle	Displayed in Layout Macros submenu and in Style pop-up menu in New Layout dialog.	Macros
LayoutMarquee	Displayed in layout marquee. This keyword is no longer recommended. See **Marquee Menu as Input Device** on page IV-151 for details.	Macros and functions
ListBoxControl	Displayed in Procedure pop-up menu in ListBox Control dialog.	Macros and functions
Panel	Displayed in Panel Macros submenu.	Macros
GizmoPlot	Displayed in Other Macros submenu.	Macros
CameraWindow	Displayed in Other Macros submenu.	Macros
FitFunc	Displayed in Function pop-up menu in Curve Fitting dialog.	Functions
ButtonControl	Displayed in Procedure pop-up menu in Button Control dialog.	Macros and functions
CheckBoxControl	Displayed in Procedure pop-up menu in Checkbox Control dialog.	Macros and functions
PopupMenuControl	Displayed in Procedure pop-up menu in PopupMenu Control dialog.	Macros and functions
SetVariableControl	Displayed in Procedure pop-up menu in SetVariable Control dialog.	Macros and functions
SliderControl	Displayed in Procedure pop-up menu in Slider Control dialog.	Macros and functions
TabControl	Displayed in Procedure pop-up menu in Tab Control dialog.	Macros and functions
CDFFunc	Displayed in the Kolmogorov-Smirnov Test dialog. See **StatsKSTest** for details.	Functions

Memory Considerations

Running out of memory is usually not an issue unless you load gigabytes of data into memory at one time. If this is true in your case, make sure to run IGOR64 (the 64-bit version of Igor) rather than IGOR32 (the 32-bit version). IGOR32 is provided only for users who rely on 32-bit XOPs that have not yet been ported to 64 bits.

On most systems, IGOR32 can access 4 GB of virtual memory. The limits for IGOR64 are much higher and depend on your operating system.

If memory becomes fragmented, you may get unexpected out-of-memory errors. This is more likely in IGOR32 than IGOR64.

See **Memory Management** on page III-451 for further information.

Wave Reference Counting

Igor uses reference counting to determine when a wave is no longer referenced anywhere and memory can be safely deallocated.

Because memory is not deallocated until all references to a given wave are gone, memory may not be freed when you think. Consider the function:

```
Function myfunc()
   Make/N=10E6 bigwave
   // do stuff
   FunctionThatKillsBigwave()
   // do more stuff
End
```

The memory allocated for bigwave is not deallocated until the function returns because Make creates an automatic wave reference variable which, by default, exists until the function returns. To free memory before the function returns, use the **WAVEClear**:

```
Function myfunc()
   Make/N=10E6 bigwave
   // do stuff
   FunctionThatKillsBigwave()
   WAVEClear bigwave
   // do more stuff
End
```

The WAVEClear command takes a list of WAVE reference variables and stores NULL into them. It is the same as:

```
WAVE/Z wavref= $""
```

Creating Igor Extensions

Igor includes a feature that allows a C or C++ programmer to extend its capabilities. Using Apple's Xcode or Microsoft Visual C++, a programmer can add command line operations and functions to Igor. These are called external operations and external functions. Experienced programmers can also add menu items, dialogs and windows.

A file containing external operations or external functions is called an "XOP file" or "XOP" (pronounced "ex-op") for short.

Here are some things for which you might want to write an XOP:

- To do number-crunching on waves.
- To import data from a file format that Igor does not support.
- To acquire data directly into Igor waves.
- To communicate with a remote computer, instrument or database.

The main reasons for writing something as an XOP instead of writing it as an Igor procedure are:

- It needs to be blazing fast.
- You already have a lot of C code that you want to reuse.
- You need to call drivers for data acquisition.
- It requires programming techniques that Igor doesn't support.
- You want to add your own dialogs or windows.

Writing an XOP that does number-crunching is considerably easier than writing a stand-alone program. You don't need to know anything about Macintosh or Windows APIs.

If you are a C or C++ programmer and would like to extend Igor with your own XOP, you need to purchase the Igor External Operations Toolkit from WaveMetrics. This toolkit contains documentation on writing XOPs as well as the source code for several WaveMetrics sample XOPs. It supplies a large library of routines that enable communication between Igor and the external code. You will also need the a recent version of Xcode, or Microsoft Visual C++.