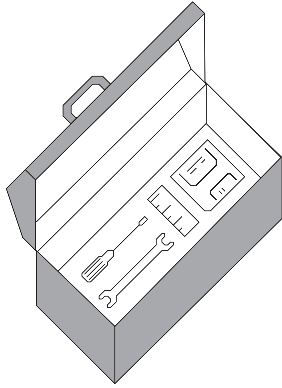


# IGOR XOP TOOLKIT

## REFERENCE MANUAL

Version 5

WaveMetrics, Inc.

## Copyright

This manual and the Igor XOP Toolkit are copyrighted by WaveMetrics with all rights reserved. Under copyright laws it is illegal for you to copy this manual or the software without written permission from WaveMetrics.

WaveMetrics gives you permission to make unlimited copies of the software on any number of machines but only for your own personal use. You may not copy the software for any other reason. You must ensure that only one copy of the software is in use at any given time.

## Warranty

WaveMetrics warrants to the registered owner that: 1) the disk on which the software is furnished will be free from defects in material and workmanship under normal use for a period of ninety (90) days from the date of delivery to you. 2) The software will be completely satisfactory to you within a period of ninety (90) days from the date of delivery to you. **WaveMetrics does not warrant, guarantee, or make any representations regarding the use or the results of the use of the software or any accompanying written materials in terms of their correctness, accuracy, reliability, currentness or otherwise. The entire risk as to the results and performance of the software and written materials is assumed by you.** (Some states do not allow the exclusion or limitation of implied warranties, so the above limitation or exclusion may not apply to you).

WaveMetrics offers a 90 day money-back guarantee on products purchased directly from us. If you are not satisfied with the product, please contact us. If we can't satisfy you, we'll refund the purchase price, not including shipping. This guarantee is also available through cooperating vendors. If you did not purchase the product directly from WaveMetrics, contact your vendor for instructions.

## Updates

WaveMetrics intends to offer periodic updates of the software to you at a reasonable price based on the new functionality added by the updates.

Please return the registration card to us so that we can let you know about updates.

If there are features that you would like to see in subsequent versions of the XOP Toolkit or if you find bugs in the current version, please let us know. We're committed to providing you with a product that does the job reliably and conveniently.

## Notice

Apple is a registered service mark of Apple Computer, Inc. Macintosh, LaserWriter, and QuickDraw are registered trademarks of Apple Computer, Inc.  Microsoft, Windows, and Visual C++ are registered trademarks of the Microsoft Corporation. CodeWarrior is a registered trademark of Metrowerks, Inc.

Manual Revision: 7/2005 (5.04)

© Copyright 1994-2005 WaveMetrics Inc. All rights reserved. Printed in the United States of America.

WaveMetrics, Inc.

PO Box 2088

Lake Oswego, OR  97035

Voice:     (503) 620-3001

FAX:     (503) 620-6754

Email:     sales@wavemetrics.com, support@wavemetrics.com

Web:     www.wavemetrics.com

# Table of Contents

Introduction to XOPs ........................................................................1
Guided Tour ........................................................................................27
Development Systems.........................................................................63
Igor/XOP Interactions.........................................................................103
Adding Operations ..............................................................................143
Adding Functions.................................................................................179
Accessing Igor Data............................................................................205
Adding Menus and Menu Items..........................................................229
Adding Windows ................................................................................247
Other Programming Topics..................................................................259
Providing Help ....................................................................................289
Debugging...........................................................................................301
XOPSupport Routines.........................................................................323
Appendix A: XOP Toolkit 5 Upgrade Notes.......................................505
Appendix B: Porting Macintosh XOPs to Carbon...............................519
Appendix C: XOP Toolkit 5 Revision History....................................539
Index ...................................................................................................545

Chapter 1

# Introduction to XOPs

What is an XOP? ........ 3
Who Can Write an XOP? ........ 4
A Brief History of Igor ........ 4
XOP Toolkit 5 ........ 5
Macintosh and Windows XOPs ........ 6
Macintosh CFM Versus Mach-O ........ 6
XOP Name Conventions ........ 7
Development Systems ........ 8
The Igor Extensions Folder On Macintosh ........ 9
Installing the XOP Toolkit ........ 10
    Testing the Macintosh Installation ........ 11
    Testing the Windows Installation ........ 12
XOP Toolkit Overview ........ 13
    The XOPSupport Folder ........ 13
    Sample XOP Folders ........ 15
XOPSupport ........ 16
The Sample XOPs ........ 17
    XOP1 ........ 17
    XFUNC1 ........ 17
    XFUNC2 ........ 17
    XFUNC3 ........ 17
    WindowXOP1 ........ 17
    MenuXOP1 ........ 18
    SimpleLoadWave ........ 18
    GBLoadWave ........ 18
    SimpleFit ........ 18
    WaveAccess ........ 19
    TUDemo ........ 19
    VDT2 (“Very Dumb Terminal”) ........ 19

NIGPIB2 ........ 19
How Igor Integrates XOPs ........ 20
The Basic Structure of an XOP ........ 22
Preparing to Write an XOP ........ 24
Technical Support ........ 26
Email Support ........ 26
FTP Support ........ 26
Igor Mailing List ........ 26
World-Wide Web ........ 26
Telephone Support ........ 26

## About This Manual

If you are new to XOP programming, we recommend that you start by reading the first four chapters of this manual. Chapter 2 is a Guided Tour. Chapter 3 deals with several development systems. You need to read only the section that pertains you your development system. After reading these chapters, you can select which of the remaining chapters you need to read depending on what your XOP is required to do.

If you have used previous versions of the XOP Toolkit, we recommend that you read this chapter and then Appendix A, which describes changes in XOP Toolkit 5. As explained in Appendix A, we recommend that you leave your existing XOP project folders as they are and create new copies of these folders for further development using XOP Toolkit 5.

If you want to port an existing Macintosh XOP that was written to run with the pre-Carbon version of Igor Pro, read Appendix B.

Appendix C describes changes to XOP Toolkit 5 since release 5.00.

## What is an XOP?

An XOP is a relatively small piece of code that extends Igor Pro.

XOPs can be very simple or quite elaborate. A simple XOP might, for example, add one operation to Igor that applies a transformation to a wave. An elaborate XOP can add multiple operations, functions, menu items, dialogs and windows. It is even possible to build a data acquisition and analysis system on top of Igor.

“XOP” literally means “external operation”. Originally XOPs were intended only to allow adding operations to Igor. Now an XOP can add much more so “XOP” has the meaning “external module that extends Igor”.

Igor has two types of built-in routines: operations and functions. An operation (e.g., Display) has an effect but no direct return value. A function (e.g., sin) has a direct return value and, usually, no side effects. An XOP can add operations and/or functions to Igor Pro.

To create an XOP, you start with sample XOP source code supplied by WaveMetrics. After modifying the sample, you compile it to produce the executable XOP.

An XOP contains executable code, required resources, and optional resources. The required resources tell Igor what operations and functions the XOP adds. The optional resources contain things such as menu items, menus, error messages, windows, and dialogs.

When Igor Pro starts up, it looks for XOPs, as well as aliases or shortcuts that point to XOPs, in the Igor Extensions folder. For each XOP that it finds, it adds the operations, functions and menu items specified by the XOP's resources. When the user accesses an operation, function or menu item added by the XOP, Igor then communicates with the XOP using a simple protocol. The XOP can access Igor data and functions using a set of routines, called XOPSupport routines, provided with the XOP Toolkit.

## Who Can Write an XOP?

In order to write an XOP you need to be very familiar with Igor. You should be very comfortable with the concepts of waves, command line operations, procedures and experiments.

To write a simple XOP you need to be comfortable with the language and development system that you are using. You will need to study and understand the simple sample XOPs supplied by WaveMetrics and the parts of this manual that apply to simple XOPs.

To write an elaborate XOP you also need to be an experienced programmer. You should start by writing a simple XOP. You will need to study and understand all of this manual.

## A Brief History of Igor

Igor 1.0 was introduced in January of 1989 and ran on Macintosh computers which, at that time, used Motorola 680x0 processors (68K). This version of Igor did not support XOPs.

WaveMetrics released Igor 1.1 in March of 1989. Igor 1.1 supported external operations but not external functions. The first XOP Toolkit was also released at this time and supported development under MPW (Macintosh Programmer's Workshop) and Symantec THINK C. Igor 1.2 shipped in May of 1990 and also supported external operations but not external functions.

In March of 1994, WaveMetrics released Igor Pro 2.0, a major new version of Igor. This version added notebooks, control panels, drawing tools, and many other features, including external functions. Shortly thereafter, WaveMetrics released XOP Toolkit 2.0, which supported XOP development under MPW and THINK C.

In May of 1995, WaveMetrics released Igor Pro 2.02, the first version optimized for Power Macintosh and its PowerPC processor (PPC). A version of the XOP Toolkit was released which supported development of PowerPC XOPs using MPW or the CodeWarrior development system from Metrowerks.

In February of 1996, WaveMetrics released Igor Pro 3.0. This version added multi-dimensional waves, text waves and data folders. XOP Toolkit 3.0 supported 68K and PPC development using MPW, THINK C, and CodeWarrior.

In November of 1997, WaveMetrics released Igor Pro 3.1. The main new thing in this version is that it ran under Windows 95 and Window NT 4.0 on Intel x86 processors as well as on Macintosh. XOP Toolkit 3.1 was released which supported development using CodeWarrior for 68K, PPC, and x86 development or Microsoft Visual C++ 5 for x86 development. Support for MPW and THINK C was dropped.

In September of 2000, WaveMetrics released Igor Pro 4.0. No new version of the XOP Toolkit was released.

In February of 2002, WaveMetrics released Igor Pro 4.05. This release included the Igor Pro Carbon application, the first version to run native on Mac OS X. It was based on Apple's Carbon API which supports software running on both Mac OS 9 and Mac OS X. Igor Pro 4.05 also included the pre-Carbon version of the application which ran on Mac OS 9 only.

In order to run native under Mac OS X, XOPs had to be revised to run under the Carbon API. In September of 2001, during the beta testing of Igor Pro Carbon, WaveMetrics released a Carbon XOP Toolkit which was shipped as part of XOP Toolkit 3.12.

In November of 2003, WaveMetrics shipped Igor Pro 5.0. For Macintosh, this release included just the Carbon version of Igor, not the pre-Carbon version. This means that any Macintosh XOP that is to run with Igor Pro 5 must be revised to use Apple's Carbon API. The Windows version of Igor Pro 5.0 requires Windows 98, Windows ME, Windows 2000 or Windows XP and does not run under Windows 95 or Windows NT.

From the XOP point of view, the major addition in Igor Pro 5 is a feature called "Operation Handler" – code within Igor that simplifies the implementation of external operations and makes it possible to call them directly from an Igor Pro user function.

In January of 2004, WaveMetrics released XOP Toolkit 5.

# XOP Toolkit 5

On Macintosh, XOP Toolkit 5 supports development of XOPs for Igor Pro 4 Carbon or Igor Pro 5. It does not support development for pre-Carbon versions of Igor. The supported development systems are Metrowerks' CodeWarrior Pro 8.3 and Apple's Xcode. The supported operating systems are Mac OS 9.1 or later or Mac OS X 10.2 or later.

On Windows, XOP Toolkit 5 supports development of XOPs for Igor Pro 4 or later. The supported development systems are Microsoft Visual C++ 6 and Microsoft Visual C++ 7 (better known as Visual C++ .NET). The supported operating systems are Windows 98, Windows ME, Windows 2000 and Windows XP.

The current version of Igor Pro is 5.00. It is possible to write XOPs that support older versions, but this adds complexity and requires additional testing. WaveMetrics recommends that you write your XOP for Igor Pro 5 only, if at all possible. If you have previously written an XOP that is used with Igor Pro 4, the recommended approach is to freeze the Igor Pro 4 version of your XOP and target new development for Igor Pro 5.
If you want to support only Igor Pro 5 then you can use any of the features and routines described in this manual. If you want to support earlier versions of Igor then you must check which version of Igor is running and restrict yourself to using only those features and routines that the running Igor supports. The section **Igor/XOP Compatibility Issues** on page 140 explains how you can check this.

If you have written XOPs with earlier versions of the XOP Toolkit, see **XOP Toolkit 5 Upgrade Notes** on page 505.

If you want to update an existing Macintosh XOP that was written to run with the pre-Carbon version of Igor Pro, read Porting Macintosh XOPs to Carbon on page 519.

# Macintosh and Windows XOPs

XOP Toolkit 5 supports development of Macintosh Carbon XOPs and Windows x86 XOPs.

Some XOPs, by their nature, are processor- and platform-independent. For example, the code for a number-crunching XOP will be nearly identical on all platforms. XOPs that have a user interface (menus, dialogs, windows) or deal with files are partially platform-dependent. However, the XOP Toolkit provides routines for dealing with menus, dialogs, windows and files. Using these routines, it is possible to write most XOPs in a mostly platform-independent way.

On Windows, the XOP Toolkit provides a bit of Macintosh emulation, mostly in the areas of memory management and menu handling. This emulation permits Windows XOPs to mesh with Igor and also makes writing XOPs more platform-independent.

# Macintosh CFM Versus Mach-O

On Mac OS X there are two binary formats for executable files: CFM (Code Fragment Manager) and Mach-O. CFM (also called PEF) is the original executable format for Power Macintosh.

Mach-O comes from the Unix underpinnings of Mac OS X. CFM executables can run on Mac OS 9 or Mac OS X. Mach-O executables run on Mac OS X only.

This table summarizes the features of the CFM and Mach-O binary executable formats.

| Format | Develop With | Runs On | Packaging | Notes |
|---|---|---|---|---|
| **CFM** | CodeWarrior | Mac OS 9 or Mac OS X | XOP is packaged as a single file. | Required if XOP is to run on Mac OS 9. |
| **Mach-O** | CodeWarrior or Xcode | Mac OS X | XOP is packaged in a folder as a “packaged bundle”. | Recommended for use with Mach-O libraries or frameworks. |

The Igor Pro application itself uses the CFM format. Prior to Igor Pro 5, XOPs also had to use the CFM format because that is what Igor understood. With Igor Pro 5, it is possible to create a Mach-O XOP for use on Mac OS X. Mach-O XOPs can not run on Mac OS 9.

There are two main reasons why you might want to create a Mach-O XOP instead of using CFM. First, Apple's Xcode development system can not create CFM executables. This means that you must use CodeWarrior to create CFM XOPs. CodeWarrior is a good development system but if you are a casual programmer you might prefer to use the free Xcode system.

The other reason for creating a Mach-O XOP is if you need to use a Mach-O library or framework, such as a Mac OS X device driver. While there are methods for calling Mach-O libraries or frameworks from CFM executables, in most cases these methods are unacceptably tedious.

A Mach-O XOP is created as a “packaged bundle”. “Bundle” is Apple's term for a plug-in type of executable. A “package” is a folder containing a particular hierarchy of subfolders and files. Although it is a folder, the Finder normally displays a package to make it look like a file. The details of XOP packaged bundles are explained in Chapter 3.

## XOP Name Conventions

Executable XOP files should have the file name extension “.xop”. This is required on Windows. For Macintosh CFM XOPs, it is optional but is recommended.

Macintosh Mach-O XOP packaged bundles must include “.xop” at the end of the package folder’s name. The executable XOP file inside the package must *not* use the “.xop” extension.

# Development Systems

This table lists the development systems supported by XOP Toolkit 5.

| Development System | Operating System | Notes |
|---|---|---|
| CodeWarrior Pro 8.3 | Mac OS 9 or Mac OS X | Earlier versions of CodeWarrior do not work correctly on Mac OS X. |
| Xcode 1.1 | Mac OS X only | Xcode is supplied with Apple’s Mac OS X developer tools. It requires Mac OS X 10.3 or later. |
| Visual C++ 6 | Windows | The "standard" (i.e., "cheaper") version of Visual C++ is fine for XOP programming. |
| Visual C++ 7 (.NET) | | |

Details on using each development system are provided in Chapter 3.

If you are an experienced programmer, you may be able to port the samples and the XOPSupport library to other development systems. You will need to read Chapter 3 to understand how XOP projects are constructed.

As new development system versions are released, WaveMetrics creates updated XOP Toolkit files and makes them available via anonymous FTP. For a list of WaveMetrics FTP sites, please see:

http://www.wavemetrics.com/support/ftpinfo.html

Most of the sample XOPs are in written in C. Many XOPs have been written in C++ also.

# The Igor Extensions Folder On Macintosh

Prior to Igor Pro 4.05, there was just one Macintosh version of Igor Pro. Active XOPs were stored in the “Igor Extensions” folder, inside the “Igor Pro Folder”:

```
Igor Pro Folder
    Igor Extensions                 // ← Active extensions stored here.
```

Starting with Igor Pro 4.05, the Macintosh version of Igor included both a Carbon version and a pre-Carbon version. The Carbon Igor application file was named “Igor Pro Carbon” while the pre-Carbon application file was named “Igor Pro”. Since XOPs needed to be ported to Carbon to work with Igor Pro Carbon, WaveMetrics had to ship two sets of XOPs. Consequently the folder hierarchy was extended, like this:

```
Igor Pro Folder
    Igor Extensions                 // ← Active pre-Carbon extensions stored here.
    Carbon Extensions and Support
        Igor Extensions             // ← Active Carbon extensions stored here.
```

So, active extensions were always stored in “Igor Extensions”, but *Carbon* active extensions were stored in a *different* “Igor Extensions” folder, nested inside the “Carbon Extensions and Support” folder.

As of Igor Pro 5, there is just one version of Igor on Macintosh again – the Carbon version. So there is no need for two Igor Extensions folders. Consequently, Igor Pro 5 uses the original organization:

```
Igor Pro Folder
    Igor Extensions                 // ← Active Carbon extensions stored here.
```

**NOTE**: When this manual refers to the Igor Extensions folder, it means the the folder containing active *Carbon* extensions. If you are running Igor Pro 5, this is “Igor Pro Folder:Igor Extensions” but if you are running Igor Pro 4, it is “Igor Pro Folder:Carbon Extensions and Support:Igor Extensions”.

## Installing the XOP Toolkit

The XOP Toolkit is delivered either via electronic download or on CD-ROM. In either case, you receive a file named XOPToolkit5.sit (*Macintosh*) or XOPToolkit5.exe (*Windows*).

To install on Macintosh, unstuff the XOPToolkit5.sit file, storing the resulting “XOP Toolkit 5” folder anywhere on your hard disk.

To install on Windows, double-click the XOPToolkit5.exe file, storing the resulting “XOP Toolkit 5” folder anywhere on your hard disk.

All of the sample XOPs, the XOPSupport library and other support files are stored in a folder named IgorXOPs5 inside the “XOP Toolkit 5” folder.

## Testing the Macintosh Installation

**If you are using CodeWarrior Pro**:

1. In the Finder, open the IgorXOPs5:XFUNC1:CW8 folder.
2. Double-click the XFUNC1.mcp project file. CodeWarrior Pro should open the project.
3. Choose Make from the CodeWarrior Project menu. This creates the XFUNC1.xop file in the IgorXOPs5:XFUNC1:CW8 folder. Don't worry if you got some warnings from the compiler during the build.
4. Make an alias from the XFUNC1.xop file and drag the alias into your Igor Extensions folder.
5. Launch Igor Pro.
6. Execute the following command:

   ```
   Print XFUNC1Add(3,4)
   ```
7. Igor should print "7" in the history area.
8. If Igor displayed an error message, you may have multiple versions of Igor on your machine. Double-check that you put the alias for the XFUNC1.xop file in the correct Igor Extensions folder. Also see **The Igor Extensions Folder On Macintosh** on page 9.

**If you are using Xcode**:

1. In the Finder, open the IgorXOPs5:XFUNC1:Xcode folder.
2. Double-click the XFUNC1.xcode project package. Xcode should open the project.
3. Choose Build from the Xcode Build menu. This creates the XFUNC1.xop package in the IgorXOPs5:XFUNC1:Xcode:build folder. Don't worry if you got some warnings from the compiler during the build.
4. Make an alias from the XFUNC1.xop package and drag the alias into your Igor Extensions folder.
4. Launch Igor Pro.
6. Execute the following command:

   ```
   Print XFUNC1Add(3,4)
   ```
7. Igor should print "7" in the history area.
8. If Igor displayed an error message, you may have multiple versions of Igor on your machine. Double-check that you put the alias for the XFUNC1.xop file in the correct Igor Extensions folder. Also see **The Igor Extensions Folder On Macintosh** on page 9.

## Testing the Windows Installation

**If you are using Visual C++ 6**:

1. In the Windows desktop, open the IgorXOPs5\XFUNC1\VC6 folder.
2. Double-click the XFUNC1.dsw file. Visual C++ 6 should open the project.
3. Choose Build XFUNC1.xop from the Build menu. This creates the XFUNC1.xop file in the IgorXOPs5\XFUNC1\VC6 folder. Don’t worry if you got some warnings from the compiler during the build.
4. Make a shortcut from the XFUNC1.xop file and drag the shortcut into your Igor Extensions folder.
5. Launch Igor Pro.
6. Execute the following command:

   ```
   Print XFUNC1Add(3,4)
   ```
7. Igor should print "7" in the history area.
8. If Igor displayed an error message, you may have multiple versions of Igor on your machine. Double-check that you put the shortcut for the XFUNC1.xop file in the Igor Extensions folder.

**If you are using Visual C++ 7 (.NET)**:

1. In the Windows desktop, open the IgorXOPs5\XFUNC1\VC7 folder.
2. Double-click the XFUNC1.sln file. Visual C++ 7 (.NET) should open the project.
3. Choose Build XFUNC1 from the Build menu. This creates the XFUNC1.xop file in the IgorXOPs5\XFUNC1\VC7 folder. Don’t worry if you got some warnings from the compiler during the build.
4. Make a shortcut from the XFUNC1.xop file and drag the shortcut into your Igor Extensions folder.
5. Launch Igor Pro.
6. Execute the following command:

   ```
   Print XFUNC1Add(3,4)
   ```
7. Igor should print "7" in the history area.
8. If Igor displayed an error message, you may have multiple versions of Igor on your machine. Double-check that you put the shortcut for the XFUNC1.xop file in the Igor Extensions folder.

# XOP Toolkit Overview

The XOP Toolkit includes a number of files that you will need to implement your XOP. The files are organized into folders all of which are in the IgorXOPs5 folder.

The IgorXOPs5 folder contains one folder for each of the sample XOPs, and an additional folder for the XOPSupport files, which are used by all XOPs.

## The XOPSupport Folder

This XOPSupport folder contains

- Compiled XOPSupport libraries for all of the supported development systems.
- The C files and headers used to make those libraries.
- For Windows, the IGOR.lib file.
- Some miscellaneous files.

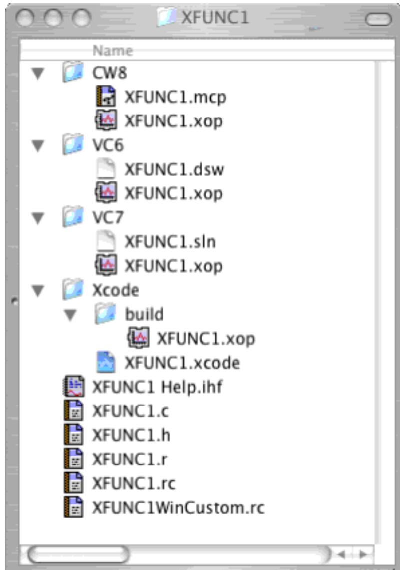
The main files of interest are listed in the following table. Your interaction with them will be mostly through including the XOPStandardHeaders.h file, linking with the library files, and calling the XOPSupport routines defined in the libraries. These routines are described in detail in Chapter 13.

| File | What It Contains |
|---|---|
| XOPSupport.c | Utility and callback routines for communication with Igor. |
| XOPWaveAccess.c | Utility and callback routines for dealing with waves. |
| XOPStandardHeaders.h | Includes ANSI-C headers, Macintosh or Windows headers, and XOP headers. All XOP C files need to include this file. |
| XOPSupport.h | Declarations and prototypes for all XOPSupport routines. This file is included in XOPStandardHeaders.h. |
| XOP.h | #defines, data structures and global variable declarations used as part of the Igor/XOP communication protocol. This file is included in XOPStandardHeaders.h. |
| IgorXOP.h | #defines for Igor error messages, menu items, number type codes, and other items culled from Igor's own source code. This file is included in XOPStandardHeaders.h. |
| :CW8: XOPSupport CFM.Lib | The CodeWarrior XOPSupport library for CFM XOPs (Mac OS 9 or Mac OS X). |
| :CW8: XOPSupport Mach.Lib | The CodeWarrior XOPSupport library for Mach-O XOPs (Mac OS X). |
| :Xcode: libXOPSupport.a | The Xcode XOPSupport library for Mac OS X XOPs. |
| \VC6\ XOPSupport.lib | The Visual C++ 6 XOPSupport library for Windows XOPs. |
| \VC7\ XOPSupport.lib | The Visual C++ 7 (.NET) XOPSupport library for Windows XOPs. |
| IGOR.lib | Allows a Windows XOP to call certain routines that are implemented in the Igor.exe file on Windows. |

In addition, the XOPSupport folder contains various header files and C files that deal with data folders, numeric conversions, menus, dialogs, windows, FIFOs, and file I/O.

## Sample XOP Folders

Each sample XOP folder contains files that are used by all development systems and files that are development-system-specific. The development-system-specific files are grouped in their own folders. For example, here is a view of the XFUNC1 folder (simplified for clarity):




Each sample XOP folder contains:

| File | Example |
|---|---|
| A C file containing the C code for the XOP. | XFUNC1.c |
| A header file containing #defines and other declarations. | XFUNC1.h |
| A resource file containing a text description of the XOP's resources. | XFUNC1.r (*Macintosh*) or XFUNC1.rc (*Windows*) |
| A "custom" resource file containing a text description of XOP-specific resources (*Windows* only) | XFUNC1WinCustom.rc |
| A resource header file containing defines used in the main resource file (*Windows* only) | resource.h |
| A help file. This is an Igor Pro notebook that has been formatted and opened in Igor Pro as a help file. | XFUNC1 Help.ihf |

| File | Example |
|---|---|
| A "project", "workspace" or "solution" file.<br>Although the terminology depends on the development system, you can think of all of these as "project" files. | XFUNC1.mcp (CodeWarrior), XFUNC1.xcode (Xcode), XFUNC1.dsw (Visual C++ 6), XFUNC1.sln (Visual C++ 7) |

When you compile an XOP, the compiler generates executable code from the C file. It then creates resources from the resource file. It writes the executable code and resources to the output executable XOP file (or to the package in the case of Mac OS X Mach-O XOPs). The projects are configured to store the compiled XOP in the project-specific folder (CW8, Xcode, VC6 or VC7).

# XOPSupport

The XOPSupport folder contains the source code for the heart of the XOP Toolkit. These routines are used by all XOPs and are described in detail in Chapter 13. Most XOPs will use just a small fraction of the available routines. The routines are made available to the XOP by including XOPStandardHeaders.h and by linking with the XOPSupport library. The name of the XOPSupport library depends on which development system you are using.

*Callbacks* are routines that request services from Igor. There are callbacks for creating and manipulating waves and variables and for many other purposes.

*Utilities* are routines that provide useful services without calling Igor. There are utilities for handling dialogs, menus, resources, files and other things.

Whether a particular XOPSupport routine is a callback or a utility is usually of no consequence to the XOP programmer.

Most callback routines are defined in the XOPSupport C files. They transmit parameters from the XOP to Igor and results from Igor back to the XOP. On Windows only, some callback routines call Igor directly. These routines are made available to the XOP by linking with IGOR.lib. The callbacks that go directly to Igor have to do with Windows-specific functions (e.g., SendWinMessageToIgor) or with Macintosh emulation (e.g., NewHandle). Whether a particular callback goes through XOPSupport or goes directly to Igor is usually of no consequence to the XOP programmer.

# The Sample XOPs

## XOP1

XOP1 is the archetypal XOP. It adds a single operation to Igor. The name of the operation is also XOP1. The XOP1 operation expects the name of a single wave. It adds the number 1 to each point in the wave. The wave must be either single or double-precision floating point.

XOP1 is a good starting point for very simple XOPs that add command line operations to Igor.

You should read XOP1.c in its entirety. It is very short – less than three pages. It illustrates the structure used by all XOPs.

## XFUNC1

XFUNC1 is a simple example of adding numeric functions to Igor Pro. It adds the XFUNC1Add(n1,n2) and XFUNC1Div(n1,n2) functions.

XFUNC1 is a good starting point for simple XOPs that add numeric functions to Igor Pro.

## XFUNC2

XFUNC2 is a more complex example of adding numeric functions to Igor Pro. It adds the logfit(w,x) and plgndr(l,m,x) functions.

logfit is a function that can be used for curve fitting. It also illustrates passing a wave to an external function.

plgndr computes Legendre polynomials.

## XFUNC3

XFUNC3 is a simple example of adding string functions to Igor Pro. It adds the xstrcat0(s1, s2) and xstrcat1(s1, s2) functions.

XFUNC3 illustrates receiving string parameters from Igor and returning string results.

## WindowXOP1

WindowXOP1 is a simple XOP which adds a window to Igor. WindowXOP1 adds a menu item to Igor's Misc menu. When you select that menu item it displays its window. WindowXOP1 shows how you can open, close, activate and update a window and how you respond to a variety of window events.

## MenuXOP1

MenuXOP1 is a simple XOP that illustrates the various ways that an XOP can add menus and menu items to Igor. It adds its own menu to Igor's main menu bar. This menu contains several submenus. It also adds a menu item to the Misc menu and a menu item with a submenu to the Analysis menu. MenuXOP1 also adds command line operations for enabling and disabling menu items and showing and hiding menus to illustrate how this is done. These operations are documented in the MenuXOP1.c file.

## SimpleLoadWave

SimpleLoadWave is a simple file loader. It loads data from tab-delimited text files into Igor waves and is a good starting point for XOPs that import data into Igor. It adds one menu item and one operation to Igor. The menu item, Load Simple Delimited File, appears in Igor's Load Waves submenu. When the menu item is chosen, the SimpleLoadWave XOP puts up an open file dialog allowing the user to select a plain text file to open. Then SimpleLoadWave opens the file, creates Igor waves and fills the waves with values from the text file. The XOP also adds the SimpleLoadWave operation to Igor. The user can invoke this operation from Igor's command line or from an Igor procedure to load tab-delimited text files.

SimpleLoadWave uses the file I/O routines provided by the XOPSupport library to achieve platform-independence.

## GBLoadWave

GBLoadWave loads general binary files into Igor waves. It adds a menu item, Load General Binary File, to Igor's Load Waves submenu. When the user chooses Load General Binary File, the GBLoadWave XOP puts up a standard Igor-style dialog which allows the user to compose a GBLoadWave command. The GBLoadWave command, when executed, creates Igor waves and fills the waves with values from the selected general binary file.

GBLoadWave is capable of reading file data of any Igor numeric type (single and double-precision floating point; 8, 16 and 32 bit signed integer; 8, 16 and 32 bit unsigned integer) and of creating waves of any type.

GBLoadWave is a good starting point for XOPs that import binary data to Igor. It also provides a good example of creating an Igor-style dialog in a mostly platform-independent way.

## SimpleFit

SimpleFit adds a simple curve-fitting function that fits to a polynomial. The Guided Tour chapter of this manual shows how to compile SimpleFit and how to change it to fit a different function.

## WaveAccess

WaveAccess illustrates three methods of accessing numeric wave data. One of the methods is optimized for speed but requires that you treat each numeric type separately. The other two methods are very easy to use with any numeric type and provide sufficient speed for most applications. WaveAccess also illustrates accessing Igor Pro text waves.

## TUDemo

TUDemo creates a simple text window in which the user can edit text. It illustrates the XOP Toolkit TU (“text utility”) routines which make it easy to implement such a window. Your XOP could use a text window to get input from the user or to display status or results.

## VDT2 (“Very Dumb Terminal”)

VDT2 is an elaborate XOP that adds a dumb terminal emulator and command line operations for serial I/O to Igor as well as a submenu in Igor's Misc menu, a dialog and a window.

VDT2 is the successor to the old VDT XOP. The main difference is that VDT2 uses Igor Pro 5’s “Operation Handler” feature so that its operations can be called from user functions as well as from macros. Prior to Igor Pro 5, external operations could not be called directly from user functions.

The VDT2 window is a text window implemented using the XOPSupport TU (“text utility”) routines. When the user chooses VDT2 Settings from VDT2’s submenu, VDT2 displays a dialog which allows the user to select the baud rate, serial port and other parameters. It stores these settings in Igor Preferences and in experiment files. VDT2 supports sending and receiving text files via a serial port. It also allows the user to send or receive Igor waves and variables.

## NIGPIB2

NIGPIB2 adds support for National Instruments GPIB cards to Igor. It adds no menu items, dialogs or windows but does add several powerful command line operations for controlling the GPIB and for transferring ASCII and binary data. NIGPIB2 is a good starting point for an XOP that interfaces Igor to hardware.

NIGPIB2 is the successor to the old NIGPIB XOP. The main difference is that NIGPIB2 uses Igor Pro 5’s “Operation Handler” feature so that its operations can be called from user functions as well as from macros. Prior to Igor Pro 5, external operations could not be called directly from user functions.

To compile NIGPIB2, you need some files from National Instruments. This is described in the file "Compiling NIGPIB2.txt".

# How Igor Integrates XOPs

When Igor Pro is launched, it searches the Igor Extensions folder and any sub-folders for XOPs or for aliases (*Macintosh*) or shortcuts (*Windows*) that point to XOPs.

Macintosh CFM XOP are identified by their file type (IXOP). Macintosh Mach-O XOPs are identified by the ".xop" extension in the name of the XOP package folder. Windows XOP files are identified by their ".xop" file name extension.

Igor first looks for an XOP's 'XOPI' ("XOP Information") resource which contains general information about the XOP. If this is missing, Igor displays an error message.

Next Igor looks for optional resources to determine what menus, menu items, operations and functions the XOP adds. Igor adds the menus to its main menu bar, adds menu items to its built-in menus, adds the operations to its list of operations and adds the functions to its list of functions.

If an XOP adds functions, Igor loads the XOP into memory at launch time and keeps it there. If the XOP adds no functions then Igor does not load it until it is needed.

The user can access an XOP by selecting one of its menu items or by invoking one of its operations or functions. The XOP then communicates with Igor using the XOP protocol.

The XOP protocol is set up in a way that simplifies coding the XOP. At appropriate times, Igor sends a message to the XOP by calling its XOPEntry function. The XOP performs the action dictated by the message or ignores the message if it is not applicable.

Every time Igor calls the XOP it passes a handle containing an IORec structure which holds all of the information that Igor and the XOP need to communicate. This handle, called an IORecHandle, is the sole parameter and is included in every call from Igor to the XOP. The message telling the XOP what it needs to do is a field in the IORec structure. You do not need to access the IORec structure directly. XOPSupport routines do this for you. The IORecHandle, like all handles passed between Igor and the XOP, is a Macintosh-style handle, even when running on Windows. For a discussion of Macintosh-style handles, see **Data Sharing** on page 139.

XOP messages fall into several logical groups. The first message that every XOP receives is the INIT message. If the user chooses an XOP's menu item, Igor sends the MENUITEM message to the XOP. If the user invokes an operation that the XOP added, Igor sends the CMD message. If the user invokes a function that the XOP added, Igor sends the FUNCTION message. There is a group of messages for XOPs which add windows to Igor. There is another group of messages for XOPs with a text window. There are messages that allow the XOP to store its settings in the current Igor experiment file or to load settings from the experiment file. Finally, when the XOP is

about to be closed, Igor sends it the CLEANUP message. The XOP can close its windows or do any other cleanup.

If your XOP is simple you can ignore most of these messages. The simplest XOP would ignore all but the INIT and CMD or FUNCTION messages.

With three exceptions, Igor always communicates with your XOP by passing a message to your XOPEntry routine. The first exception is the INIT message. Igor sends this by calling your main routine. The second exception is the FUNCTION message. For optimum speed, you can instruct Igor to call your external function directly rather than by passing the FUNCTION message to your XOPEntry routine. Details on this are in Chapter 6. The third exception is the CMD message. External operations that use Igor's Operation Handler feature (described in Chapter 5) are called directly rather than by passing the CMD message to your XOPEntry routine.

Once the XOP has received an applicable message it needs to do something. Igor provides plenty of help. An XOP can call Igor back, requesting a service. This is called a "callback". When the XOP does a callback, it passes the IORecHandle back to Igor. This time the handle contains a message for Igor requesting a service. You don't need to explicitly deal with the IORecHandle since the XOPSupport routines do it for you.

On Windows only, a few callbacks, mostly pertaining to memory management and menus, go directly to Igor and do not use the IORecHandle.

Like XOP messages, callback messages fall into several categories. There are callbacks to access Igor waves and variables. There are callbacks to open a text window and to handle all of the things the user might do in the text window. There is a callback that allows an XOP to put a message (called a "notice") in Igor's history window. There is also a callback that allows an XOP to execute a standard Igor command.

A simple XOP may use just a few callbacks.

XOPs can be *transient* or *resident*. A transient XOP is one that is invoked by the user, executes and is then purged from memory. A resident XOP is one that needs to remain in memory indefinitely. An XOP that adds a window or that controls an on-going process needs to be resident. If an XOP adds a function or an operation that uses Operation Handler then it also must be resident.

You can control whether your XOP is transient or resident using the SetXOPType XOPSupport routine. Because most XOPs add either a function or an operation that uses Operation Handler, most XOPs will be resident.

# The Basic Structure of an XOP

All XOP programs have the same simple structure which is illustrated in the sample XOPs. A message from Igor is the trigger for an XOP action.

The first message that your XOP receives is the INIT message. This message is received by your main routine. It must do two kinds of initialization: initialization common to all XOPs and initialization specific to your XOP.

After the INIT message, most messages are handled by your XOPEntry routine. It figures out what the message is and responds appropriately. For simple XOPs the response to most messages is to do nothing because the message does not apply to the XOP. When a message does apply to your XOP, you must respond. Often the response will be to do the appropriate callback to Igor.

External operations and functions are usually called directly, rather than going through the XOPEntry routine.

In very rough terms, you can think of your XOP program as follows:

```
// Igor calls this for the corresponding external function
ExternalFunction()
{
   Process parameters
   Return result
}

// Igor calls this for the corresponding external operation
ExternalOperation()
{
   Process parameters
   Return result
}

XOPEntry()           // Igor sends most messages here
{
   Receive message from Igor
   switch (message)
      Respond to message (often includes a callback to Igor)
   }
}

main()               // Igor sends the INIT message here
{
   initialize
}
```

Here is a more detailed sketch of a basic XOP program. In this example, the XOP adds an external operation and a menu item to Igor.

```
//  XOP.c -- A sample Igor external operation

#include "XOPStandardHeaders.h"   // Include ANSI-C, Mac or Win headers,XOP headers
#includes specific to your XOP go here

// Global Variables
Declaration of any global variables specific to this XOP

static int
RegisterOperation()               // Register external operation with Igor
{
}

static int
ExecuteOperation()                // Service external operation
{
    Process parameters
    Return result
}

static int
MenuItem()                        // Handle selection of XOP's menu items if any
{
    Determine which menu item the user selected
    switch (menuItem) {
        Handle all menu items
    }
}

static void
XOPQuit()
{
    Do any necessary cleanup before the XOP is closed.
}

static void
XOPEntry()                        // Called by Igor for all XOP messages after INIT.
{
    switch (GetXOPMessage()) {    // What message did we receive?
        case MENUITEM:            // The user selected the XOP's menu item
            MenuItem();
            break;

        Other cases here as needed

        case CLEANUP:
            XOPQuit();
            break;
    }
}

void
main(IORecHandle ioRecHandle)     // First call from Igor goes here.
{
    XOPInit(ioRecHandle);         // Initialization common to all XOPs
    SetXOPEntry(XOPEntry);        // Set entry point for future messages

    XOP specific initialization
    RegisterOperation();          // Register external operation with Operation Handler
}
```

An XOP that adds just a command line operation will be simpler than this sketch since it will have no routines for dealing with menus.

An XOP that adds just a function will have an ExecuteFunction routine in place of ExecuteOperation.

Notice that Igor passes a parameter of type IORecHandle to the XOP's main function. This handle contains all of the information that Igor needs to communicate with the XOP.

The XOPInit(ioRecHandle) and SetXOPEntry(XOPEntry) calls in main are very important. Both of the called routines are defined in XOPSupport.c.

XOPInit stores the ioRecHandle in a global variable for use by the other XOPSupport routines. This is the last time that you have to deal with the ioRecHandle directly. From here on, the routines in XOPSupport deal with it for you.

SetXOPEntry sets a field in the ioRecHandle that tells Igor the address of the function in the XOP to pass future messages to. XOPEntry is the function in every XOP's main .c file that handles messages from Igor.

XOPs that add a window to Igor are somewhat more complex. On Macintosh, all window-related messages (e.g., clicks, typing) come to the XOPEntry routine from Igor. On Windows, it is different. The Windows OS sends window-related messages directly to the XOP window's window procedure. See Chapter 9 for details.

# Preparing to Write an XOP

Before you write your XOP, you need to have a feel for several things. You can get this feel by a combination of doing the Guided Tour in Chapter 2, playing with the sample XOPs, examining their source code and reading this manual.

You should understand the idea of Igor passing a message to an XOP. You should understand the idea of the XOP responding, including using callbacks to implement the response. You should understand the specific messages and callbacks used by a simple XOP.

You need to know the mechanics of compiling an XOP in your development system. This is covered in Chapter 3. Play with a sample XOP. Get used to the cycle of compiling, linking and testing the XOP. Then make a trivial modification to it. Compile, link and test that.

Once you've got the feel for these things you can start to write your XOP. Identify the messages and callbacks that you will need to implement your XOP. Identify the sample XOP that is the best starting point for your XOP. Modify this sample XOP little by little, until you've reached your goal.

In most cases you must quit Igor to recompile your XOP.

Here are some other topics that you might need to know about before you diverge too far from the sample XOPs.

Read **Data Sharing** on page 139 and **Macintosh Memory Management** on page 261 for information on memory management techniques.

If your XOP adds operations, functions or menu items to Igor, it might need to display its own error messages. See the section **XOP Errors** on page 127.

If your XOP adds an operation to Igor, read Chapter 5, **Adding Operations**.

If your XOP adds a function to Igor, read Chapter 6, **Adding Functions** .

If your XOP needs to manipulate waves, variables or data folders, read Chapter 7, **Accessing Igor Data**.

If your XOP adds menus or menu items to Igor, read Chapter 8, **Adding Menus and Menu Items**.

If your XOP adds a window to Igor, read Chapter 9, **Adding Windows**.

If your menu items summon dialogs, read **Adding Dialogs** on page 269.

If your XOP saves its settings or documents as part of an Igor experiment, read **XOPs and Experiments** on page 132.

To create a help file for your XOP, read Chapter 11, **Providing Help**.

Chapter 12 lists common pitfalls.

Chapter 12 also includes a discussion of the most common sources of bugs, ways to find them, and ways to avoid them in the first place. Reading this chapter will probably save you a lot of time and aggravation.

# Technical Support

WaveMetrics provides technical support via telephone and email. Before contacting us, you might want to check the following sources of information:

- The list of common pitfalls in Chapter 12
- The sample XOPs

Before contacting WaveMetrics, please gather this information so that we can help you more effectively:

- The exact version of Igor you are running. The version number is displayed in the About Igor Pro dialog.
- On Macintosh, the exact version of the Macintosh operating system you are running.
- On Windows, the operating system that you are using (Windows 98, ME, 2000, XP).
- The development system that you are using.

## Email Support

You can send questions to us via email at `support@wavemetrics.com`.

For sales matters, please use `sales@wavemetrics.com`.

## FTP Support

For a list of WaveMetrics FTP sites, please see::

```
http://www.wavemetrics.com/support/ftpinfo.html
```

## Igor Mailing List

There is a mailing list of Igor users on the Internet. This is the best way to keep up with the latest Igor news. To join the list or search archives from the list, see:

```
http://www.wavemetrics.com/MailingList/igorList.html
```

## World-Wide Web

Our Web address is `www.wavemetrics.com`.

## Telephone Support

You can reach us at (503) 620-3001 from 9 AM to 5 PM Pacific time.

It is often very helpful if you can try things on your computer while speaking to us so, if possible, call us from a phone near your computer.

*Chapter* 2

# Guided Tour

Overview ........................................ 29
What We Will Do ........................................ 29
Installation ........................................ 29
Building SimpleFit ........................................ 30
Creating a New Project ........................................ 35
    Creating the New Project In CodeWarrior ........................................ 36
    Creating the New Project In Xcode ........................................ 41
    Creating the New Project In Visual C++ 6 ........................................ 45
    Creating the New Project In Visual C++ 7 (.NET) ........................................ 49
Changing the Resources ........................................ 53
    Changing the Resources in CodeWarrior ........................................ 53
    Changing the Resources in Xcode ........................................ 53
    Changing the Resources in Visual C++ 6 ........................................ 54
    Changing the Resources in Visual C++ 7 ........................................ 55
Changing the Help ........................................ 56
Changing and Compiling the Code ........................................ 58
Testing SimpleGaussFit ........................................ 60
Where To Go From Here ........................................ 62

# Overview

This guided tour gives a step-by-step example of creating a custom curve fitting external function using the SimpleFit project as a starting point. The tour consists of explanation interspersed with numbered steps for you to do.

The tour gives instructions for Metrowerks CodeWarrior Pro 8.3, Apple's Xcode 1.1, Microsoft Visual C++ 6 and Microsoft Visual C++ 7 (.NET). If you are using a newer development system, you may need to make adjustments but the basic process remains the same.

The tour assumes that you are running Igor Pro 5. It will work with Igor Pro 4 but on Macintosh you must use Igor Pro Carbon and keep in mind that the term "Igor Extensions" refers to the Carbon version of the Igor Extensions folder as described on page 9.

Where the instructions are different for the different development systems, you will see steps identified as follows.

**1-CodeWarrior.**

<Step for CodeWarrior 8.3>.

**1-Xcode.**

<Step for Xcode 1.1>.

**1-Visual C++ 6.**

<Step for Visual C++ 6>.

**1-Visual C++ 7.**

<Step for Visual C++ 7 (.NET)>.

During the tour, you will be instructed to restart Igor Pro several times because this is necessary when you recompile an external function.

# What We Will Do

The SimpleFit project creates a single SimpleFit external function that can be used to fit to a polynomial. We will convert SimpleFit into a function called SimpleGaussFit that fits to series of Gaussian peaks. In doing this, you will see how to create an external function that fits to a function of your choice.

# Installation

We assume you have already installed your development system, Igor Pro, and the XOP Toolkit.

## Building SimpleFit

Before starting the modification of SimpleFit, we will build the original version to verify that the compiler and XOP toolkit installations are correct.

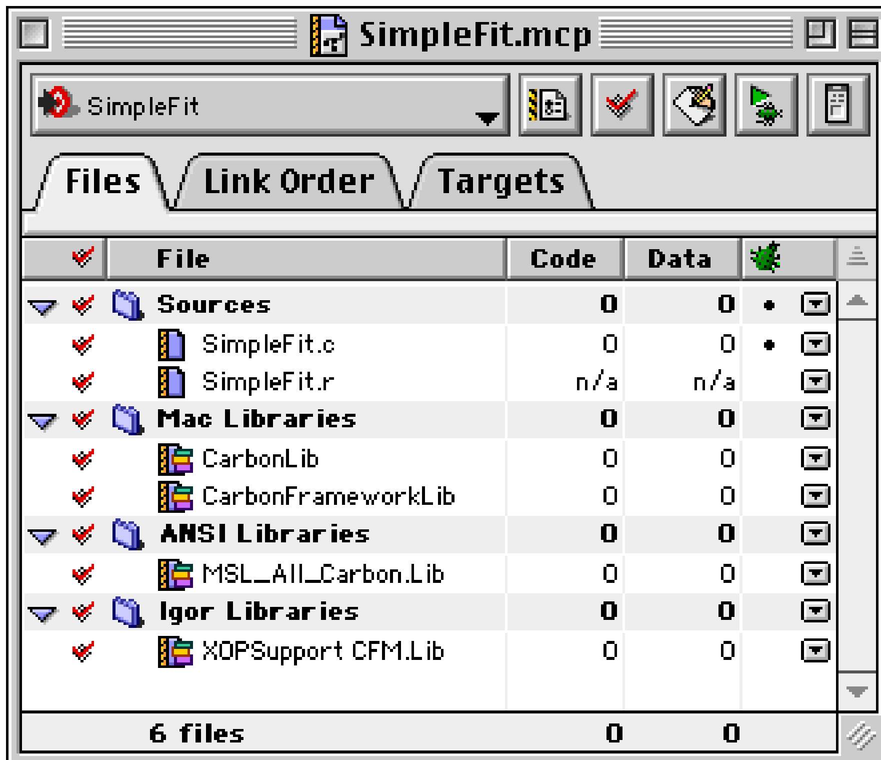
The SimpleFit folder is inside the IgorXOPs5 folder and contains the following project files:

| | |
|---|---|
| :CW8:SimpleFit.mcp | For CodeWarrior 8.3 on Mac OS 9 or Mac OS X. |
| :Xcode:SimpleFit.xcode | For Xcode 1.1 on Mac OS X 10.3 or later |
| \VC6\SimpleFit.dsw | For Visual C++ 6 on Windows |
| \VC7\SimpleFit.sln | For Visual C++ 7 (.NET) on Windows |

**1-CodeWarrior.**

**Double-click the SimpleFit.mcp project file in SimpleFit:CW8.**

Metrowerks CodeWarrior starts up and the project window is shown.



**2-CodeWarrior.**

**Choose Project->Make.**

In a few seconds the compiler will finish and an XOP file named SimpleFit.xop will be created in your SimpleFit:CW8 folder. Note where this is located for use in a upcoming step.

Don't worry if the compiler issues a few warnings.

**3-CodeWarrior.**

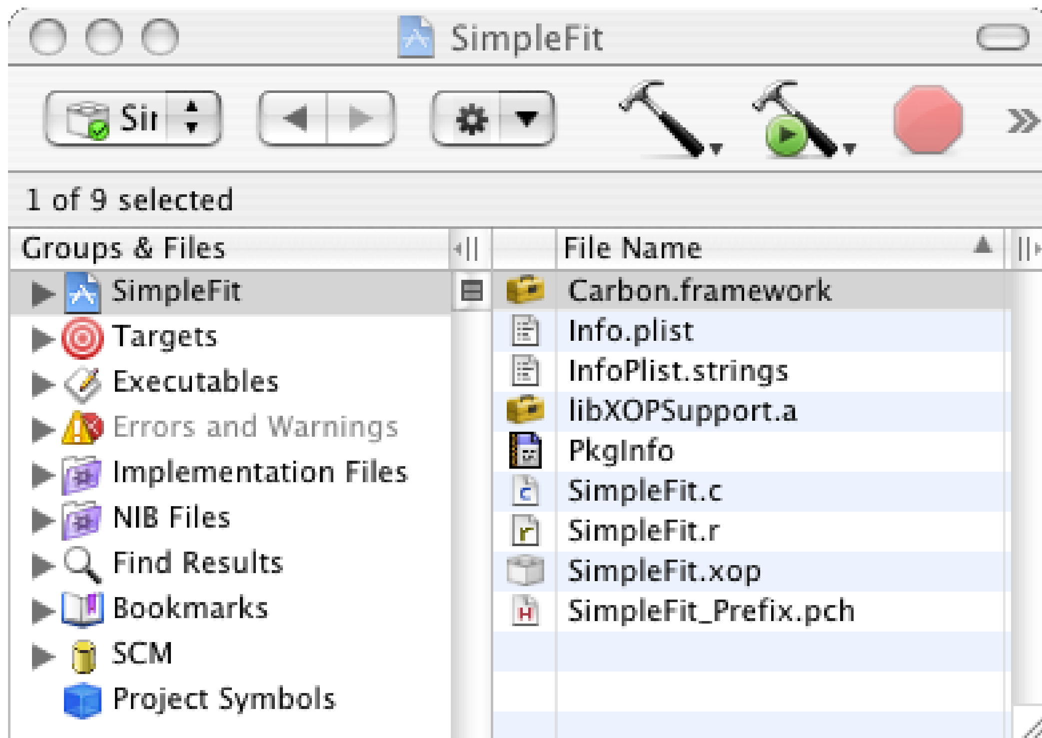
**Close the project window.**

We are done with this project now.

**1-Xcode.**

**Double-click the SimpleFit.xcode project file in SimpleFit:Xcode.**

Xcode starts up and the project window is shown.



**2-Xcode.**

**Choose Build->Build menu.**

In a few seconds the compiler will finish and an XOP package named SimpleFit.xop will be created in your SimpleFit:Xcode:build folder. Note where this is located for use in a upcoming step.

Don't worry if the compiler issues a few warnings.

**3-Xcode.**

**Close the project window.**

We are done with this project now.

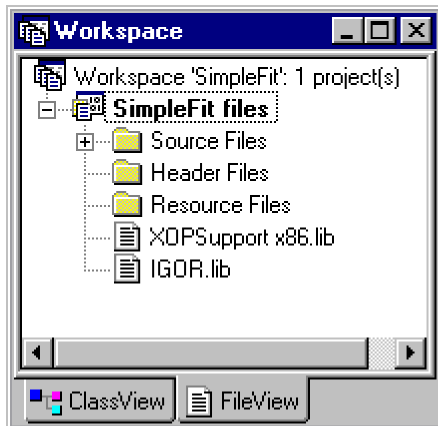
**1-Visual C++ 6.**

**Double-click the SimpleFit.dsw project workspace file in SimpleFit\VC6.**

Visual C++ starts up and the Workspace window is shown.

**Click the FileView tab in the Workspace window and open the SimpleFit Files icon.**

The window should now look like this:




**2-Visual C++ 6.**

**Choose "Build SimpleFit.xop" from the Build menu.**

In a few seconds the compiler will finish and an XOP file named SimpleFit.xop will be created in your SimpleFit\VC6 folder. Note where this is located for use in a upcoming step.

Don't worry if the compiler issues a few warnings.

**3-Visual C++ 6.**

**Choose Close Workspace from the File menu.**

We are done with this project now.

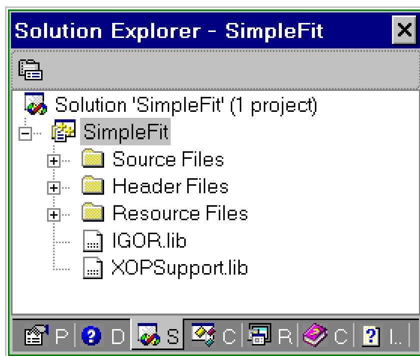
**1-Visual C++ 7 (.NET).**

**Double-click the SimpleFit.sln solution file in SimpleFit\VC7.**

Visual C++ starts up.

**Choose View–>Solution Explorer and open the SimpleFit icon.**

The window should now look like this:




**2-Visual C++ 7 (.NET).**

**Choose Build Solution from the Build menu.**

In a few seconds the compiler will finish and an XOP file named SimpleFit.xop will be created in your SimpleFit\VC7 folder. Note where this is located for use in a upcoming step.

Don’t worry if the compiler issues a few warnings.

**3-Visual C++ 7 (.NET).**

**Choose Close Solution from the File menu.**

We are done with this project now.

4. **In the desktop, make an alias (*Macintosh*) or shortcut (*Windows*) for the newly created SimpleFit.xop and drag the alias or shortcut to the Igor Extensions folder in your Igor Pro Folder.**

   You could drag the XOP itself instead of the alias or shortcut. We prefer to leave the XOP in the project folder while we are working on it.

5. **Back in the SimpleFit folder, create an alias (*Macintosh*) or shortcut (*Windows*) for the "SimpleFit Help.ihf" file and put the alias or shortcut into the folder containing the just-compiled XOP. Make sure the alias or shortcut has the exact same name as the help file itself.**

   Igor looks for the help file in the folder containing the XOP itself (SimpleFit.xop in this case). Normally the help file and the XOP reside in the same folder but this is not true during development. With Igor Pro 5.02 or later, Igor will find the help file if you put an identically-named alias or shortcut to it in the same folder as the XOP. Prior to Igor Pro 5.02, you had to move the help file itself.

6. **Launch Igor Pro.**

   You must restart Igor Pro after changing the contents of the Igor Extensions folder.

7. **Choose Command Help from the Help menu.**

8. **Select External from the Functions popup menu above the list and uncheck the Operations and Programming checkboxes.**

9. **Find SimpleFit in the list and select it.**

   The custom help for this external function is shown. This help is stored in the "SimpleFit Help.ihf" file which is in the same folder as the SimpleFit XOP.

   If you are using Xcode, you need Igor Pro version 5.01 or later to see the help.

10. **Follow the instructions in the help for exercising the function.**

    This involves copying commands from the help and executing them in the Igor command line. You can also execute commands from a help file by selecting the command text and pressing Control-Enter (*Macintosh*) or Ctrl-Return (*Windows*).

# Creating a New Project

We now create a new project, SimpleGaussFit, by cloning the SimpleFit project. We will then change the SimpleGaussFit project to fit a Gaussian.

1. **On the desktop, duplicate the SimpleFit folder and rename the new folder SimpleGaussFit.**

   On Macintosh, use Duplicate in the Finder's File menu.

   On Windows, use copy and paste in the desktop.

2. **On the desktop, in the SimpleGaussFit folder, delete the CW8, VC6, VC7 and Xcode folders.**

   Later you will recreate the folder appropriate for your development system.

3. **Change all file and folder names starting with "SimpleFit" to start with "SimpleGaussFit".**

   The remaining files are:

   resource.h
   SimpleGaussFit Help.ihf
   SimpleGaussFit.c
   SimpleGaussFit.r
   SimpleGaussFit.rc
   SimpleGaussFitWinCustom.rc

From the next four sections, choose the one that is appropriate for your development system.

## Creating the New Project In CodeWarrior

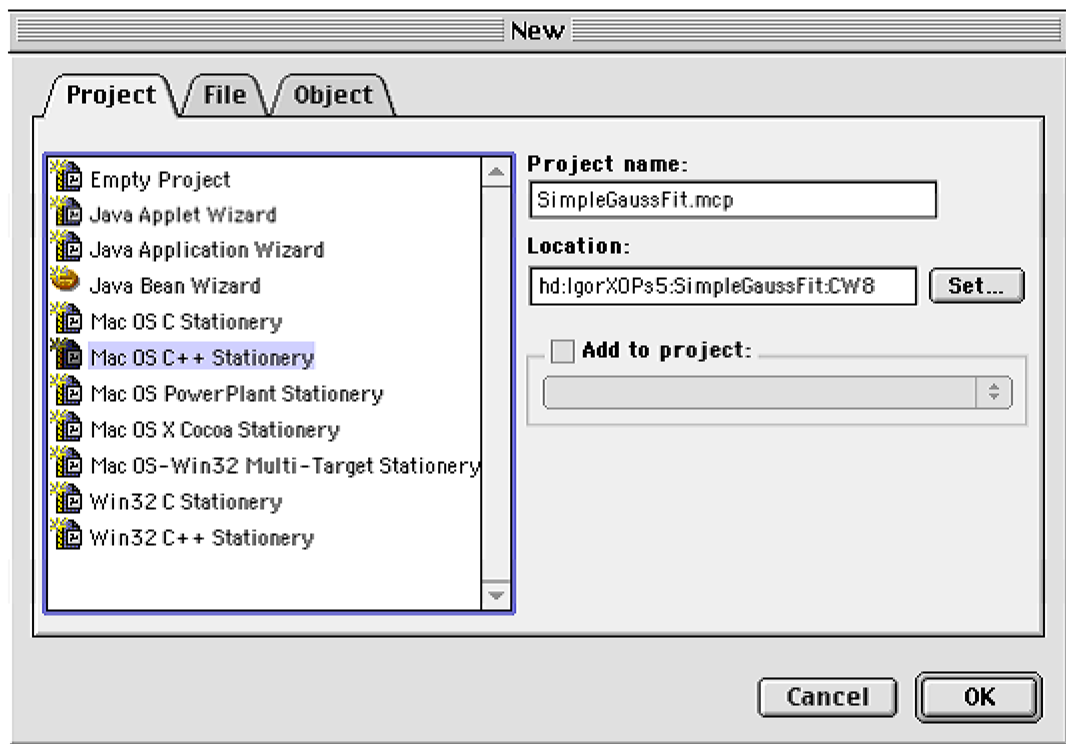
**1-CodeWarrior.**

**On the desktop, create a folder named CW8 inside the SimpleGaussFit folder.**

This folder will hold files specific to the CodeWarrior development system.

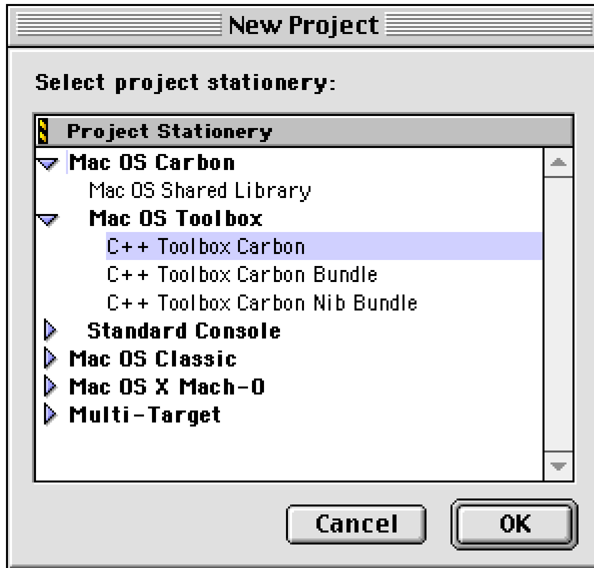
**2-CodeWarrior.**

**In CodeWarrior, choose File->New. CodeWarrior will display the dialog shown below. Set it up as shown, but adjust the Location based on where you put your IgorXOPs5 folder.**



**3-CodeWarrior.**

**Click the OK button. The New Project window will appear. Set it up like this:**




Although we are not using C++, we create a C++ project in case we want to use C++ in the future. The main upshot of choosing C++ here is that the C++ compiler will be used instead of the C compiler and the C++ compiler is more picky about syntax.

**4-CodeWarrior.**

**Click the OK button.**

CodeWarrior will create a new project inside the :SimpleGaussFit:CW8 folder.

**5-CodeWarrior.**

**In the desktop, verify that you have a SimpleGaussFit.mcp CodeWarrior project file inside :IgorXOPs:CW8.**

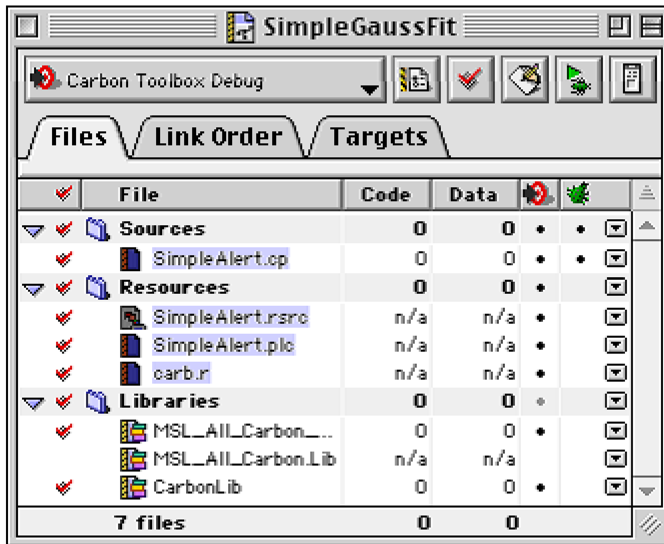
If this is not what you have, close the CodeWarrior project window and go back to step 1.

**6-CodeWarrior.**

**In the CodeWarrior project window, open the group icons and select the files shown below.**

**Now control-click and choose Clear from the resulting popup menu to remove the files from the project.**

These are sample CodeWarrior files that we do not need.




**7-CodeWarrior.**

**On the desktop, delete the sample CodeWarrior files from the SimpleGaussFit:CW8 folder:**

carb.r
SimpleAlert.cp
SimpleAlert.plc
SimpleAlert.rsrc

**8-CodeWarrior.**

**In the CodeWarrior project window, click the Sources icon, choose Project->Add Files and add the following files:**

SimpleGaussFit.c
SimpleGaussFit.r

CodeWarrior will ask you which targets you want to add the files to. Add them to all targets.

CodeWarrior will display a window saying that it has created an access path to the files.

**9-CodeWarrior.**

**In the CodeWarrior project window, click the Libraries icon, choose Project->Add Files and add the following file:**

:IgorXOPs5:XOPSupport:CW8:XOPSupport CFM.lib

CodeWarrior will ask you which targets you want to add the library to. Add it to all targets.

CodeWarrior will display a window saying that it has created an access path to the library.

**10-CodeWarrior.**

**In the CodeWarrior, choose Edit->Carbon Toolbox Debug Settings and make the following changes to the target settings:**

**Target**

***Target Settings***

*Target Name*: SimpleGaussFit Debug

***Access Paths***

Click in the User Paths section to activate it.

Click the Add button and add the IgorXOPs5:XOPSupport folder as a project-relative access path.

The resulting path should be displayed as: {Project}:::XOPSupport:

***Runtime Settings***

Click the Choose button and find your Igor Pro 5 application file.

***PPC Target***

*Project*: Shared Library

*File Name*: SimpleGaussFit.xop

*Creator*: IGR0

*Type*: IXOP

**Linker**

***PPC Linker***

*Initialization*: __initialize (starts with two underscores)

*Main*: main

*Termination*: __terminate (starts with two underscores)

In some versions of CodeWarrior, failure to enter the initialization and termination entry points as shown caused a crash when a C++ XOP threw an exception.

**11-CodeWarrior.**

**Click the Save button to save the target settings and then close the settings window.**

**12-CodeWarrior.**

**Choose Project->Make to make CodeWarrior compile the XOP.**

**13-CodeWarrior.**

**In the desktop, verify that you have an XOP file named SimpleGaussFit.xop in IgorXOPs5/SimpleGaussFit/CW8.**

**14-CodeWarrior.**

**Make an alias for the SimpleGaussFit.xop XOP file in and put the alias in your Igor Extensions folder.**

## Creating the New Project In Xcode

If your Xcode version (as shown in the About Xcode dialog) is less than 1.1, you should update to the latest Xcode. These instructions were created with Xcode 1.1 and are slightly different from instructions for version 1.01.

**1-Xcode.**

**On the desktop, create a folder named Xcode inside the SimpleGaussFit folder.**

This folder will hold files specific to the Xcode development system.

**2-Xcode.**

**Choose File->New Project.**

**Select Bundle, Carbon Bundle, click Next.**

**Enter SimpleGaussFit as the project name.**

**Click the Choose button. In the resulting dialog, select your IgorXOPs5/SimpleGaussFit/Xcode folder and click the Choose button.**

The Location edit box should now contain a path like:

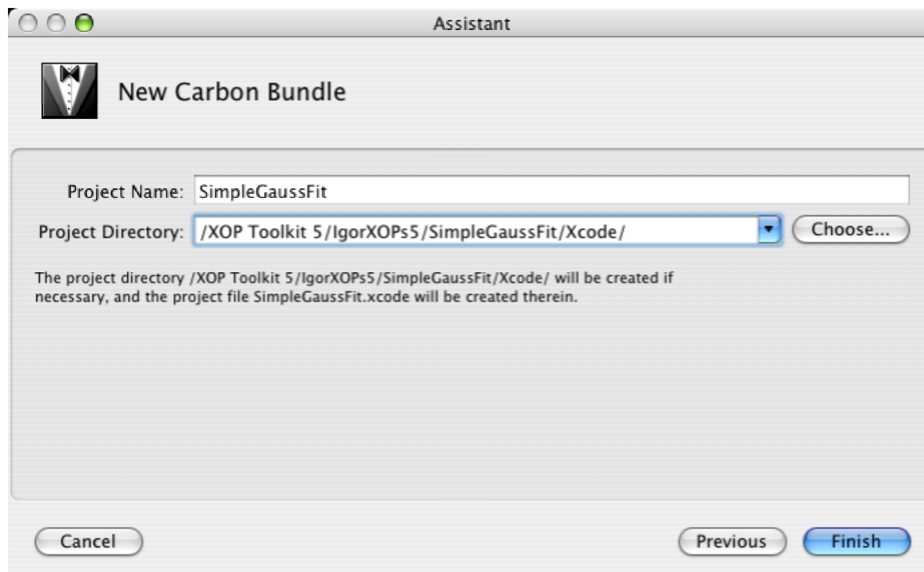
/XOP Toolkit 5/IgorXOPs5/SimpleGaussFit/Xcode/SimpleGaussFit/

This is ***not*** what we want.

**Remove the last part to give:**

/XOP Toolkit 5/IgorXOPs5/SimpleGaussFit/Xcode/

The window should now look something like this:



The Project Directory box will show something different depending on where you put your IgorXOPs5 folder. But make sure that the path points to the Xcode folder that you created in the SimpleGaussFit folder.

**3-Xcode.**

**Click Finish.**

This puts some project files in the Xcode folder. SimpleGaussFit.xcode is the project package. In the Finder verify that you now have a hierarchy something like this:

```
SimpleGaussFit
    Xcode
        build
        SimpleGaussFit.xcode
```

There will be other files and folders as well as those shown above.

**4-Xcode.**

**In Xcode, in the project window, open the SimpleGaussFit group at the top of the Groups & Files list, then open the Source folder in the SimpleGaussFit group.**

**5-Xcode.**

**Select the main.c sample file that was put there by Xcode.**

**Choose Edit->Delete.**

**In the resulting dialog, click the Delete References and Files button.**

This removes main.c from the project and deletes it from disk.

**6-Xcode.**

**Click the Source group icon and choose Project->Add Files.**

**In the resulting dialog, select the following files:**

SimpleGaussFit.c
SimpleGaussFit.r

**Now click the Add button.**

Xcode now presents another dialog.

**7-Xcode.**

**From the Reference Type popup menu, choose Relative to Project.**

**Click the Add button.**

We make the references project-relative so that if you move your IgorXOPs5 folder, the project references will remain valid.

**8-Xcode.**

**In the Groups & Files list, select the SimpleGaussFit->External Frameworks & Librairies icon.**

**Choose Project->Add Files. Add the file:**

IgorXOPs5/XOPSupport/Xcode/libXOPSupport.a.

Xcode now presents another dialog.

**9-Xcode.**

**From the Reference Type popup menu, choose Relative to Project.**

**Click the Add button.**

**10-Xcode.**

**In the desktop, find the Exports.exp file in the SimpleFit:Xcode folder and *copy* it to the SimpleGaussFit.Xcode folder.**

The Exports.exp file is a plain text file that tells the Xcode what symbols the project exports. In the case of an XOP, the only exported symbol is "main" since this is the only routine in the XOP that Igor must be able to locate by name.

**11-Xcode.**

**In the Groups & Files list, open the Targets icon.**

**Select the Targets->SimpleGaussFit icon and choose Project->Get Info.**

This displays the info window for the SimpleGaussFit target.

**Click the Build tab.**

**Click the Customized Settings icon in the Collections tray and enter the following build settings:**

| | |
|---|---|
| ***Header Search Paths*** | ../../XOPSupport |
| ***Library Search Paths*** | ../../XOPSupport/Xcode |
| ***Wrapper Extension*** | xop |
| ***OTHER_REZ_FLAGS*** | -i ../../XOPSupport -d TARGET_RT_MAC_MACHO |

The last letter of TARGET_RT_MAC_MACHO is "oh", not zero. If you spell it wrong, your XOP's XOPI resource will be wrong and Igor will display an error when you try to run your XOP.

**12-Xcode.**

**Open the General icon in the Collections tray, select the Linking icon, and enter this setting:**

| | |
|---|---|
| ***Exported Symbols File*** | ./Exports.exp |

**13-Xcode.**

**Select the Packaging icon, and enter this setting:**

| | |
|---|---|
| ***Force Package Info Generation*** | Checked (that is, check the checkbox) |

**14-Xcode.**

**Click the Properties tab at the top of the info window and enter the following settings:**

| | | |
|---|---|---|
| ***Type*** | IXOP | (third character is “oh”, not zero) |
| ***Creator*** | IGR0 | (last character is zero, not “oh”) |

**15-Xcode.**

**Close the info window.**

**16-Xcode.**

**Choose Build->Build.**

Xcode will build the project. You may get some warnings about items where the Xcode compiler is pickier than most. When Xcode is finished, it should display a “Build succeeded” message.

**17-Xcode.**

**In the desktop, verify that you have a package folder named SimpleGaussFit.xop in IgorXOPs5/SimpleGaussFit/Xcode/build.**

This package folder is your compiled XOP. It should look like a file in the Finder and you should have to Control-click it and choose Show Package Contents to see what it contains. At least, that's the theory. We have had problems getting it to work. If your folder looks like a folder, don’t worry – it won’t prevent the XOP from working and we’ll give instructions for fixing it in Chapter 3 (**Xcode XOP Package** on page 85.).

**18-Xcode.**

**Make an alias for the SimpleGaussFit.xop package folder and put the alias in your Igor Extensions folder.**

## Creating the New Project In Visual C++ 6

**1-Visual C++ 6.**

**On the desktop, create a folder named VC6 inside the SimpleGaussFit folder.**

This folder will hold files specific to the Visual C++ 6 development system.

**2-Visual C++ 6.**

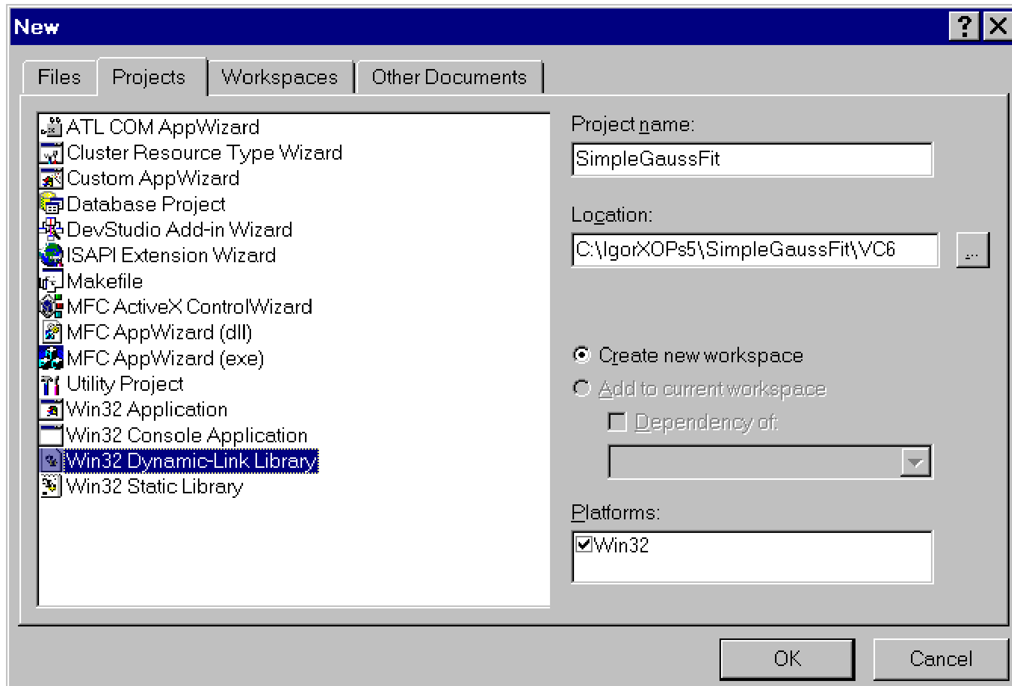
**In Visual C++ 6, choose File->New to display the New dialog.**

**Select Win32 Dynamic-Link Library.**

**Enter SimpleGaussFit as the project name.**

**In the Location box, enter the path to the VC6 folder that you just created.**

The dialog should look something like this:




The Location box will show something different depending on where you put your IgorXOPs5 folder. But make sure that the path points to the VC6 folder that you created in the SimpleGaussFit folder.

**3-Visual C++ 6.**

**Click the OK button.**

Visual C++ 6 displays a “Win32 Dynamic-Link Library” wizard.

**4-Visual C++ 6.**

**Select the radio button labeled "An empty DLL project".**

**Click the Finish button.**

Visual C++ 6 displays a New Project Information dialog.

**5-Visual C++ 6.**

**Verify the Project Directory shown at the bottom/left corner of the dialog.**

It should point to the VC6 folder that you created in the SimpleGaussFit folder. If not, click Cancel and go back and fix it.

**6-Visual C++ 6.**

**Click the OK button.**

**7-Visual C++ 6.**

**In the desktop, verify at the contents of your VC6 folder.**

It should contain files named SimpleGaussFit.dsp (the project file) and SimpleGaussFit.dsw (the workspace file).

**8-Visual C++ 6.**

**In Visual C++ 6, choose Project->Settings.**

**In the Project Settings dialog, in the Settings For menu, choose All Configurations.**

**Enter the settings shown below.**

Only things that need to be changed relative to the default settings are listed here.

***Debug Pane***

*General Category*

*Executable for debug session*: <Path to your Igor Pro executable>

(e.g., C:\Program Files\WaveMetrics\Igor Pro Folder\Igor.exe)

***C/C++ Pane***

*Code Generation Category*

*Use run-time library*: Single-Threaded

*Struct member alignment*: 2 Bytes

*Preprocessor Category*

*Additional include directories*: ..\..\XOPSupport

***Link Pane***

*General Category*

*Output file name*: SimpleGaussFit.xop

*Object/library modules*: Add version.lib

*Input Category*

*Ignore Libraries*: Add libcd.lib

***Resource Pane***

*Additional resource include directories*: ..\..\XOPSupport

**9-Visual C++ 6.**

**Click the OK button in the Project Settings dialog.**

**10-Visual C++ 6.**

**Choose Build->Set Active Configuration.**

**In the resulting dialog, select the debug configuration and click OK.**

**11-Visual C++ 6.**

**Choose Save Workspace from the File menu.**

**12-Visual C++ 6.**

**Click the FileView tab on the Workspace window and open the SimpleGaussFit Files icon.**

**Select the SimpleGaussFit Files icon.**

**13-Visual C++ 6.**

**Choose Project->Add To Project->Files and add the following files:**

SimpleGaussFit.c
SimpleGaussFit.rc

If you get an error message saying that Visual C++ can't find "afxres.h", this is because you did not install MFC (Microsoft Foundation Class) when you installed Visual C++ 6. See instructions for handling this on page 89.

**14-Visual C++ 6.**

**Choose File->Open.**

**Select Text from the Open As popup menu.**

**Open the SimpleGaussFit.rc for editing as text.**

**Find the string "SimpleFitWinCustom.rc". It occurs in two places.**

This file name is left over from the original SimpleFit XOP.

**Change both occurrences to "SimpleGaussFitWinCustom.rc".**

**Save and close the SimpleGaussFit.rc file.**

**15-Visual C++ 6.**

**Choose Project->Add To Project->Files.**

**In the resulting dialog, choose Library Files from the Files of Type menu.**

**Navigate to the IgorXOPs5\XOPSupport folder and add IGOR.lib.**

**16-Visual C++ 6.**

**Choose Project->Add To Project->Files.**

**In the resulting dialog, choose Library Files from the Files of Type menu.**

**Navigate to the IgorXOPs5\XOPSupport\VC6 folder and add XOPSupport.lib.**

**17-Visual C++ 6.**

**Choose File->Save Workspace.**

**18-Visual C++ 6.**

**Choose Build->Build SimpleGaussFit.xop.**

Visual C++ 6 should build the XOP with no errors (though you may get some warnings), creating the file SimpleGaussFit.xop in your IgorXOPs5\ SimpleGaussFit\VC6 folder.

If you get an error message saying that Visual C++ can't find "afxres.h", this is because you did not install MFC (Microsoft Foundation Class) when you installed Visual C++ 6. See instructions for handling this on page 79.

**19-Visual C++ 6.**

**Make a shortcut for the SimpleGaussFit.xop file and put the shortcut in your Igor Extensions folder.**

## Creating the New Project In Visual C++ 7 (.NET)

**1-Visual C++ 7.**

**On the desktop, create a folder named VC7 inside the SimpleGaussFit folder.**

This folder will hold files specific to the Visual C++ 7 development system.

**2-Visual C++ 7.**

**In Visual C++ 7, choose File->New->Project to display the New Project dialog.**

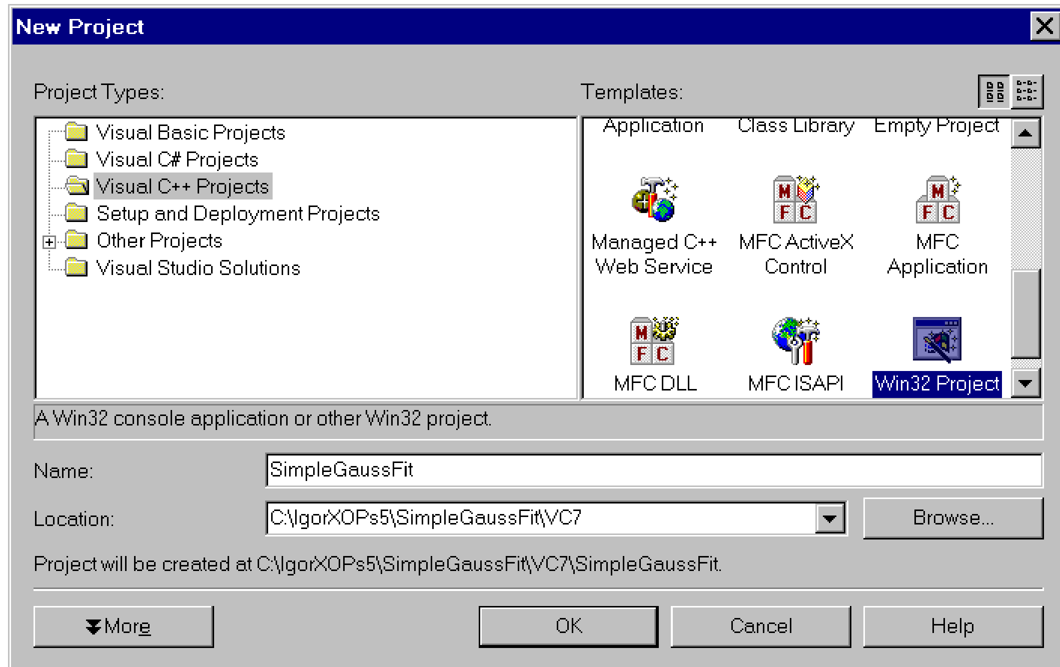
**From the Project Types list, select Visual C++ Projects.**

**From the Templates list, select Win32 Project.**

**Enter SimpleGaussFit as the project name.**

**In the Location box, enter the path to the VC7 folder that you just created.**

The dialog should look something like this:




The Location box will show something different depending on where you put your IgorXOPs5 folder. But make sure that the path points to the VC7 folder that you created in the SimpleGaussFit folder.

Note where the dialog says "Project will be created at . . .". We really want the project files to be created in the VC7 folder but Visual C++ 7 insists on creating another folder inside VC7. We will fix that later.

**3-Visual C++ 7.**

**Click the OK button.**

Visual C++ 7 displays a “Win32 Application Wizard” window.

**4-Visual C++ 7.**

**Click Application Settings.**

**Click the DLL radio button.**

**Check the Empty Project checkbox.**

**Click the Finish button.**

Visual C++ 7 creates the project files.

**5-Visual C++ 7.**

**Choose File->Close Solution.**

**6-Visual C++ 7.**

**In the desktop, open the VC7 folder that you created.**

It should contain the extra SimpleGaussFit folder that Visual C++ 7 created.

**Move the following files from the \VC7\SimpleGaussFit folder to the \VC7 folder:**
SimpleGaussFit.sln (the “solution” file)
SimpleGaussFit.vcproj (the project file)

**7-Visual C++ 7.**

**Delete the \VC7\SimpleGaussFit folder which now contains only non-essential files that Visual C++ 7 created.**

**8-Visual C++ 7.**

**Verify that you now have the following hierarchy:**

```
IgorXOPs5
    SimpleGaussFit
        VC7
            SimpleGaussFit.sln
            SimpleGaussFit.vcproj
```

**9-Visual C++ 7.**

**In Visual C++ 7, choose File->Open->Project and open the SimpleGaussFit.sln file in the VC7 folder.**

**10-Visual C++ 7.**

**Choose View->Solution Explorer.**

**In the Solution Explorer window, open the SimpleGaussFit icon.**

**11-Visual C++ 7.**

**Choose Project->Add Existing Item and add the following files:**

SimpleGaussFit.c
SimpleGaussFit.rc

**12-Visual C++ 7.**

**Choose File->Open->File.**

**Select Resource Files from the Files Of Type popup menu.**

**Select SimpleGaussFit.rc in the file list.**

**Click the down-arrow at the right edge of the Open button and choose Open With.**

**Choose "Source Code (Text) Editor".**

**Click the Open button to open SimpleGaussFit.rc for editing as text (not as a resource).**

**Find the string "SimpleFitWinCustom.rc". It occurs in two places.**

This file name is left over from the original SimpleFit XOP.

**Change both occurrences to "SimpleGaussFitWinCustom.rc".**

**Save and close the SimpleGaussFit.rc file.**

**13-Visual C++ 7.**

**Choose Project->Add Existing Item.**

**In the resulting dialog, choose All Files from the Files of Type menu.**

**Navigate to the IgorXOPs5\XOPSupport folder and add IGOR.lib.**

**14-Visual C++ 7.**

**Choose Project->Add Existing Item.**

**In the resulting dialog, choose All Files from the Files of Type menu.**

**Navigate to the IgorXOPs5\XOPSupport\VC7 folder and add XOPSupport.lib.**

**15-Visual C++ 7.**

**In Solution Explorer, right-click the SimpleGaussFit icon and choose Properties.**

**In the SimpleGaussFit Property Pages window, in the Configuration menu, choose All Configurations.**

**Enter the settings shown below.**

Only things that need to be changed relative to the default settings are listed here.

***General***

*Build Browser Information*: Yes

***C/C++***

*General*

*Additional Include Directories*: ..\..\XOPSupport

*Code Generation*

*Runtime Library*: Single-Threaded

*Struct Member Alignment*: 2 Bytes

***Linker***

*General*

*Output File*: SimpleGaussFit.xop

*Input*

*Additional Dependencies*: version.lib

***Resources***

*Additional Include Directories*: ..\..\XOPSupport

**16-Visual C++ 7.**

**Click the OK button in the SimpleGaussFit Property Pages window.**

**17-Visual C++ 7.**

**In Solution Explorer, right-click the SimpleGaussFit icon and choose Properties.**

**In the Configuration menu, choose Debug and enter the following setting:**

***Debugging***

*Command*: <Path to your Igor Pro executable>
(e.g., C:\Program Files\WaveMetrics\Igor Pro Folder\Igor.exe)

**Click the OK button.**

**18-Visual C++ 7.**

**Select the SimpleGaussFit icon in the Solution Explorer window.**

**Choose File->Save SimpleGaussFit.**

**19-Visual C++ 7.**

**Choose Build->Build SimpleGaussFit.**

Visual C++ 7 should build the XOP with no errors (though you may get some warnings), creating the file SimpleGaussFit.xop in your IgorXOPs5\ SimpleGaussFit\VC7 folder.

**20-Visual C++ 7.**

**Make a shortcut for the SimpleGaussFit.xop file and put the shortcut in your Igor Extensions folder.**

# Changing the Resources

Igor examines an XOP's resources to determine what operations, functions, and menu items the XOP adds. In this section we modify the project’s resources so they apply to SimpleGaussFit instead of SimpleFit.

## Changing the Resources in CodeWarrior

**1-CodeWarrior.**

**In the project window, double-click SimpleGaussFit.r file to edit the file.**

**2-CodeWarrior.**

**Replace all occurrences of “SimpleFit” with “SimpleGaussFit”.**

One of the things you changed was the XOPF resource. This is how Igor knows what function this XOP adds, what its parameters are and what its return value is.

**3-CodeWarrior.**

**Save and close SimpleGaussFit.r.**

## Changing the Resources in Xcode

**1-Xcode.**

**In the project window, double-click SimpleGaussFit.r file to edit the file.**

**2-Xcode.**

**Replace all occurrences of “SimpleFit” with “SimpleGaussFit”.**

One of the things you changed was the XOPF resource. This is how Igor knows what function this XOP adds, what its parameters are and what its return value is.

**3-Xcode.**

**Save and close SimpleGaussFit.r.**

## Changing the Resources in Visual C++ 6

In Visual C++, two resource files are used: SimpleGaussFit.rc and SimpleGaussFitWinCustom.rc. SimpleGaussFit.rc contains standard Windows resources, such as the version resource. SimpleGaussFitWinCustom.rc contains XOP-specific resources. Because Visual C++ allows you to directly include only one .rc file in a project, SimpleGaussFit.rc contains an include statement to include SimpleGaussFitWinCustom.rc.

In the following steps we will first modify SimpleGaussFitWinCustom.rc and then SimpleGaussFit.rc.

**1-Visual C++ 6.**

**Choose Open (not Open Workspace) from the File menu.**

**In the resulting dialog, choose Text from the Open As popup menu.**

**Open the SimpleGaussFitWinCustom.rc file.**

This opens the file for editing as a text file rather than using the resource editor.

**2-Visual C++ 6.**

**Replace all occurrences of "SimpleFit" with "SimpleGaussFit".**

One of the things you changed was the XOPF resource. This is how Igor knows what function this XOP adds, what its parameters are and what its return value is.

**3-Visual C++ 6.**

**Save and close SimpleGaussFitWinCustom.rc.**

**4-Visual C++ 6.**

**Choose Open (not Open Workspace) from the File menu.**

**In the resulting dialog, choose Text from the Open As popup menu.**

**Open the SimpleGaussFit.rc file.**

This opens the file for editing as a text file rather than using the resource editor.

**5-Visual C++ 6.**

**Replace all occurrences of "SimpleFit" with "SimpleGaussFit".**

The name occurs in the version resource and also in two include statements. However, we already changed the include statements as part of creating the project.

**6-Visual C++ 6.**

**Save and close SimpleGaussFit.rc.**

## Changing the Resources in Visual C++ 7

In Visual C++, two resource files are used: SimpleGaussFit.rc and SimpleGaussFitWinCustom.rc. SimpleGaussFit.rc contains standard Windows resources, such as the version resource. SimpleGaussFitWinCustom.rc contains XOP-specific resources. Because Visual C++ allows you to directly include only one .rc file in a project, SimpleGaussFit.rc contains an include statement to include SimpleGaussFitWinCustom.rc.

In the following steps we will first modify SimpleGaussFitWinCustom.rc and then SimpleGaussFit.rc.

**1-Visual C++ 7.**

**Choose File->Open->File.**

**Select SimpleGaussFitWinCustom.rc in the file list.**

**Click the down-arrow at the right edge of the Open button and choose Open With.**

**Choose “Source Code (Text) Editor”.**

**Click the Open button to open SimpleGaussFitWinCustom.rc for editing as text (not as a resource).**

**2-Visual C++ 7.**

**Replace all occurrences of “SimpleFit” with “SimpleGaussFit”.**

One of the things you changed was the XOPF resource. This is how Igor knows what function this XOP adds, what its parameters are and what its return value is.

**3-Visual C++ 7.**

**Save and close SimpleGaussFitWinCustom.rc.**

**4-Visual C++ 7.**

**Choose File->Open->File.**

**Select SimpleGaussFit.rc in the file list.**

**Click the down-arrow at the right edge of the Open button and choose Open With.**

**Choose “Source Code (Text) Editor”.**

**Click the Open button to open SimpleGaussFit.rc for editing as text (not as a resource).**

**5-Visual C++ 7.**

**Replace all occurrences of "SimpleFit" with "SimpleGaussFit".**

The name occurs in the version resource and also in two include statements. However, we already changed the include statements as part of creating the project.

**6-Visual C++ 7.**

**Save and close SimpleGaussFitWinCustom.rc.**

## Changing the Help

In this section, we modify the help that will appear for SimpleGaussFit in Igor Pro's Help Browser. This process consists of editing the help text in the "SimpleGaussFit Help.ihf" file.

1. **In Igor Pro, choose File->Open File->Notebook and open the "SimpleGaussFit Help.ihf" file in the SimpleGaussFit folder.**

   We are opening the help file as a notebook so we can edit it.

   On Windows, select All Files from the Files of Type popup menu to see ".ihf" files in the Open File dialog.

2. **Replace all occurrences of "SimpleFit" with "SimpleGaussFit".**

3. **Change the body of the help text to describe the SimpleGaussFit function.**

   Here is some suggested body text:

   ```
   A fitting function for multiple Gaussian peaks. The number
   of peaks is set by the length of the coefficient wave w. If
   w contains four points then one peak will be generated as
   follows: w[0] + w[1]*exp(-((x-w[2])/w[3])^2)
   Add three additional coefficients for each additional peak
   by adding three points to the coefficients wave.
   ```

   If this were a "real" project, we would provide more help with an example of how to use the external function.

4. **Save the notebook by choosing File->Save Notebook.**

5. **Close the notebook window. When Igor asks if you want to kill or hide it, click Kill.**

Now we will compile the help file so that SimpleGaussFit will provide help in Igor's Help Browser.

6. **Choose File->Open->Help File and open the “SimpleGaussFit Help.ihf” file as a help file.**

   Igor will display dialog asking if you want to compile the help file.

   **Click the Yes button.**

   Next Igor will display a dialog saying that the help file has been compiled.

   **Click the OK button.**

7. **Close the help file by pressing the option key (*Macintosh*) or the Alt key (*Windows*) while clicking the help window's close box.**

   The help will appear in Igor's Help Browser after we have compiled and activated the SimpleGaussFit XOP.

# Changing and Compiling the Code

If you were to compile the project right now you would create an external function named SimpleGaussFit but it would still act like SimpleFit. In this section, we modify the code to fit a Gaussian rather than a polynomial.

1. **In your development system, open SimpleGaussFit.c for editing.**
2. **Change all instances of "SimpleFit" to "SimpleGaussFit".**
3. **Find the SimpleGaussFit function and make the following changes:**

   **Change**

   ```
   double r,x;
   ```

   **To**

   ```
   double r,x,t;
   ```

   **Then, in the NT_FP32 case statement, change:**

   ```
   i = np-1;
   r = fp[i];
   for(i=i-1; i>=0; i--)
      r = fp[i] + r*x;
   ```

   to

   ```
   r = fp[0];
   for(i=1; i<np; i+=3) {
      t = (x-fp[i+1]) / fp[i+2];
      r += fp[i] * exp(-t*t);
   }
   ```

   ***Carefully proofread the changes.***

**Similarly, in the NT_FP64 case statement, change:**

```
i = np-1;
r = dp[i];
for(i=i-1; i>=0; i--)
   r = dp[i] + r*x;
```

**to**

```
r = dp[0];
for(i=1; i<np; i+=3) {
   t = (x-dp[i+1]) / dp[i+2];
   r += dp[i] * exp(-t*t);
}
```

***Carefully proofread the changes.***

**4A. Change the comments at the head of the function and at the top of the file such that they properly reflect the new code.**

**4B-CodeWarrior.**

**If you are using CodeWarrior, add this line near the top of the file:**

```
using std::exp;
```

This is needed because the CodeWarrior C++ header files put the standard math functions in the std namespace. Other development systems do not do that.

**5. Save your changes and close the SimpleGaussFit.c window.**

**6-CodeWarrior.**

**Compile the project by choosing Project->Make.**

The compile and link should proceed without errors (although you may get some warnings), creating SimpleGaussFit.xop in the IgorXOPS5:SimpleGaussFit:CW8 folder. If you have errors, carefully check the changes that you made to the source code.

**6-Xcode.**

**Compile the project by choosing Build->Build.**

The compile and link should proceed without errors (although you may get some warnings), creating SimpleGaussFit.xop in the IgorXOPS5:SimpleGaussFit:Xcode:build folder. If you have errors, carefully check the changes that you made to the source code.

**6-Visual C++ 6.**

**Choose "Build SimpleGaussFit.xop" from the Build menu.**

The compile and link should proceed without errors (although you may get some warnings), creating SimpleGaussFit.xop in the IgorXOPS5:SimpleGaussFit:VC6 folder. If you have errors, carefully check the changes that you made to the source code.

**6-Visual C++ 7.**

**Choose "Build SimpleGaussFit" from the Build menu.**

The compile and link should proceed without errors (although you may get some warnings), creating SimpleGaussFit.xop in the IgorXOPS5:SimpleGaussFit:VC7 folder. If you have errors, carefully check the changes that you made to the source code.

# Testing SimpleGaussFit

Your external function, SimpleGaussFit, should now be functional. Let's test it.

1. **Quit Igor Pro.**

   Igor Pro scans XOPs at launch time and loads XOPs that add external functions into memory. So you must restart Igor when you change your XOP.

2. **In the SimpleGaussFit folder, drag the "SimpleGaussFit Help.ihf" file into the folder containing the just-compiled XOP.**

   Igor looks for the help file in the folder containing the XOP itself (SimpleGaussFit.xop in this case).

3. **Restart Igor Pro.**

   Igor Pro will load the SimpleGaussFit function into memory.

4. **Choose Help->Command Help.**

5. **Select External from the Functions popup menu above the list and uncheck the Operations and Programming checkboxes.**

**6. Find SimpleGaussFit in the list and select it.**

The custom help for this external function is shown. Igor gets the help from the "SimpleGaussFit Help.ihf" file which must be in the same folder as the SimpleGaussFit XOP.

If you don't see SimpleGaussFit in the list of functions:

- Check that you created an alias (*Macintosh*) or shortcut (*Windows*) from SimpleGaussFit.xop and placed the alias or shortcut in the Igor Extensions folder.
- If you have more than one Igor Extensions folder on your machine, check that you have put the alias or shortcut in the right Igor Extensions folder. See page 9.
- Check that you put the "SimpleGaussFit Help.ihf" file in the folder containing SimpleGaussFit.xop.
- Check that you changed all occurrences of SimpleFit to SimpleGaussFit in SimpleGaussFit.c, SimpleGaussFit.r (*Macintosh*), and SimpleGaussFit.rc and SimpleGaussFitWinCustom.rc (*Windows*).

If you do see SimpleGaussFit in the list of functions but get a message saying help is not available when you select it:

- Check that you put the "SimpleGaussFit Help.ihf" file in the folder containing SimpleGaussFit.xop.
- Check that you compiled the "SimpleGaussFit Help.ihf" file as directed on page 56.
- If you are using Xcode, update to the latest Igor Pro. You need at least Igor Pro version 5.01 to see the help.

**7. Close the Help Browser.**

**8. Execute the following commands to exercise your external function with single-precision data:**

```
Make data; SetScale x,-1,10,data
data = 0.1 + gnoise(0.05)               // offset
data += 2*exp(-((x-2)/0.5)^2)           // first peak
data += 3*exp(-((x-4)/1)^2)             // second peak
Display data
Modify rgb=(0,0,50000), mode=2, lsize=3
Make coef = {0.1, 1.5, 2.2, 0.25, 3.8, 3.5, 0.5}
FuncFit SimpleGaussFit coef data /D
```

To avoid errors due to arithmetic truncation it is best to use double-precision for curve fitting.

**9. Execute the following commands to exercise your external function with double-precision data:**

```
Redimension/D data, coef
coef = {0.1, 1.5, 2.2, 0.25, 3.8, 3.5, 0.5}
FuncFit SimpleGaussFit coef data /D
```

If the fit doesn’t converge or looks incorrect, check the changes that you made to SimpleGaussFit.c and the commands that you executed in Igor.

If you need to recompile the XOP, you will need to quit Igor Pro, recompile, and relaunch Igor Pro. This is necessary because Igor Pro loads the external function code into memory at launch time.

## Where To Go From Here

Read Chapter 3 and Chapter 4 in this manual.

Look at the chapter titles in the rest of this manual to see which chapters you will need to read.

Find a sample XOP that would be a good starting point for your XOP and read that XOP's code.

*Chapter* 3

# Development Systems


Overview ..... 65
XOPs in CodeWarrior Pro ..... 66
CFM XOP Projects in CodeWarrior Pro 8 ..... 67
Target Settings ..... 69
Access Paths ..... 70
Runtime Settings ..... 70
PPC Target ..... 71
C/C++ Language ..... 71
PPC Processor ..... 71
Global Optimizations ..... 72
PPC Linker ..... 72
Debugging a CodeWarrior CFM XOP ..... 73
Mach-O XOP Projects in CodeWarrior Pro 8 ..... 75
Debugging a CodeWarrior Mach-O XOP ..... 78
XOPs in Xcode ..... 79
XOP Projects in Xcode ..... 80
Xcode Project Settings ..... 83
Xcode C++ Projects ..... 84
Xcode XOP Package ..... 85
Debugging an Xcode XOP ..... 86
Other Xcode Notes ..... 88
fopen Function ..... 88
Help Files ..... 88
Balloon Help ..... 88
Structures Defined in Parameter Lists ..... 88
XOPs in Visual C++ 6 ..... 89
XOP Projects in Visual C++ 6 ..... 90
Visual C++ 6 Project Settings ..... 92

Debug Tab/General Category ..... 92
C/C++ Tab/Code Generation Category ..... 92
C/C++ Tab/Preprocessor Category ..... 92
Link Tab/General Category ..... 92
Link Tab/Input Category ..... 92
Resources Tab ..... 92
Debugging a Visual C++ 6 XOP ..... 93
XOPs in Visual C++ 7 (.NET) ..... 94
XOP Projects in Visual C++ 7 ..... 95
Visual C++ 7 Project Settings ..... 97
Debugging Properties ..... 97
C/C++ Properties /General Category ..... 97
C/C++ Properties /Code Generation Category ..... 97
Linker Properties /General Category ..... 97
Linker Properties /Input Category ..... 97
Resources Properties ..... 97
XOPSupport Warnings ..... 98
Debugging a Visual C++ 7 XOP ..... 98
Writing XOPs in C++ ..... 99
Mixing C and C++ Code ..... 99
Code Changes For C++ ..... 99
Using C++ Exceptions ..... 100
C++ XOPs in CodeWarrior Pro ..... 101
CodeWarrior CFM Project Settings ..... 101
Using the new Operator in CodeWarrior ..... 101
C++ XOPs in Xcode ..... 102
C++ XOPs in Visual C++ 6 ..... 102
C++ XOPs in Visual C++ 7 ..... 102

# Overview

XOP Toolkit 5 includes sample XOPs and supporting files for the following development systems:

| Development System | Operating System | Vendor |
|---|---|---|
| CodeWarrior Pro 8.3 | Mac OS 9 or Mac OS X | Metrowerks Corporation |
| Xcode 1.1 | Mac OS X only | Apple Computer |
| Visual C++ 6 | Windows | Microsoft |
| Visual C++ 7 (.NET) | | |

If you are using a later development system version, keep in mind that they may differ slightly from what is described here.

On Macintosh an XOP is a shared library. On Windows it is a dynamic link library (DLL). If you are an advanced programmer, you can probably figure out how to use other development systems.

As new development system versions are released, WaveMetrics creates updated XOP Toolkit files and makes them available via anonymous FTP. For a list of WaveMetrics FTP sites, please see:

http://www.wavemetrics.com/support/ftpinfo.html

The XOP Toolkit CD-ROM also contains an older XOP Toolkit version that work with old versions of CodeWarrior as well as Visual C++ 5. These should be used only as a last resort.

# XOPs in CodeWarrior Pro

CodeWarrior Pro, published by Metrowerks Corporation, is a C and C++ development system that can build programs for Macintosh and Windows. The XOP Toolkit includes support for Macintosh development only.

XOP Toolkit 5 provides sample projects and support files for CodeWarrior Pro 8.3, the last version of CodeWarrior that runs on both Mac OS 9 and Mac OS X. These samples and support files should work in later versions of CodeWarrior, although minor tweaks may be necessary.

We recommend that you do not try to use earlier versions of CodeWarrior as earlier versions had numerous problems, especially for Mac OS X development,.

Using CodeWarrior, you can compile both CFM (Code Fragment Manager) and Mach-O XOPs. CFM and Mach-O refer to two executable file formats supported by Mac OS X. CFM is also supported by Mac OS 9. Igor Pro 4 supported CFM XOPs only while Igor Pro 5 supports both CFM and Mach-O XOPs.

If Mac OS 9-compatibility or Igor Pro 4-compatibility is required, you must create a CFM XOP. If you need to use Mac OS X frameworks (such as IOKit or hardware device drivers), you must create a Mach-O XOP. If neither of these apply then you can create either. Creating a CFM XOP is slightly simpler. For further details on CFM vs Mach-O, see page 6.

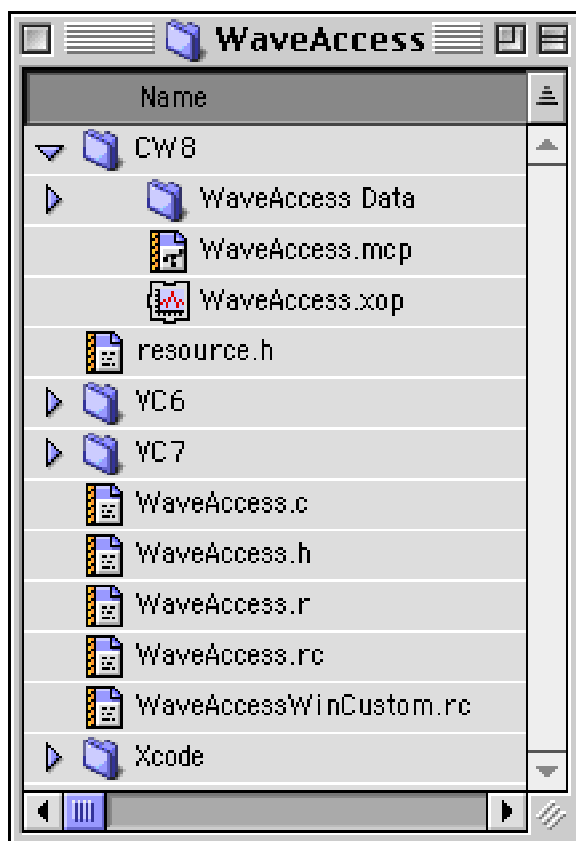
The sample projects are provided as CFM projects except for VDT2 which is a Mach-O project. Both CFM and Mach-O versions of the XOPSupport library are provided.

## CFM XOP Projects in CodeWarrior Pro 8

This section provides the background information needed to understand how to create CFM XOP projects in CodeWarrior. For step-by-step instructions on creating a project, see **Creating a New Project** on page 35 and Creating the New Project In CodeWarrior on page 36.

If you are unsure about settings for your project, you may find it handy to open your project and a WaveMetrics sample project at the same time to compare settings.

We will use the WaveAccess sample XOP as a case in point. The WaveAccess folder is inside the IgorXOPs5 folder and looks like this:



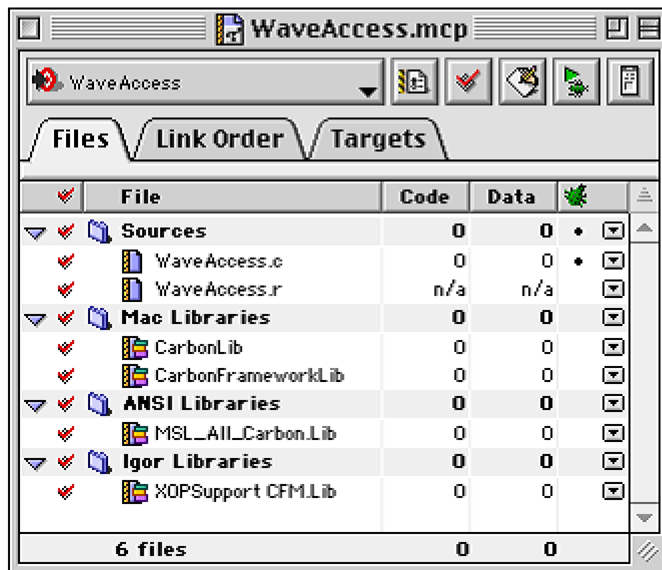

The CodeWarrior project files are inside the CW8 folder, to keep them separate from the files for the other development systems. The discussion assumes this arrangement. Note that the output XOP file is also in the CW8 folder.

The WaveAccess.mcp file is the project file and contains the project settings. The WaveAccess Data folder contains data created by the compiler such as object code.

WaveAccess.c contains the project source code while WaveAccess.h contains the project headers.

WaveAccess.r contains the XOP's Macintosh resources. The files resource.h, WaveAccess.rc and WaveAccessWinCustom.rc are used on Windows only.

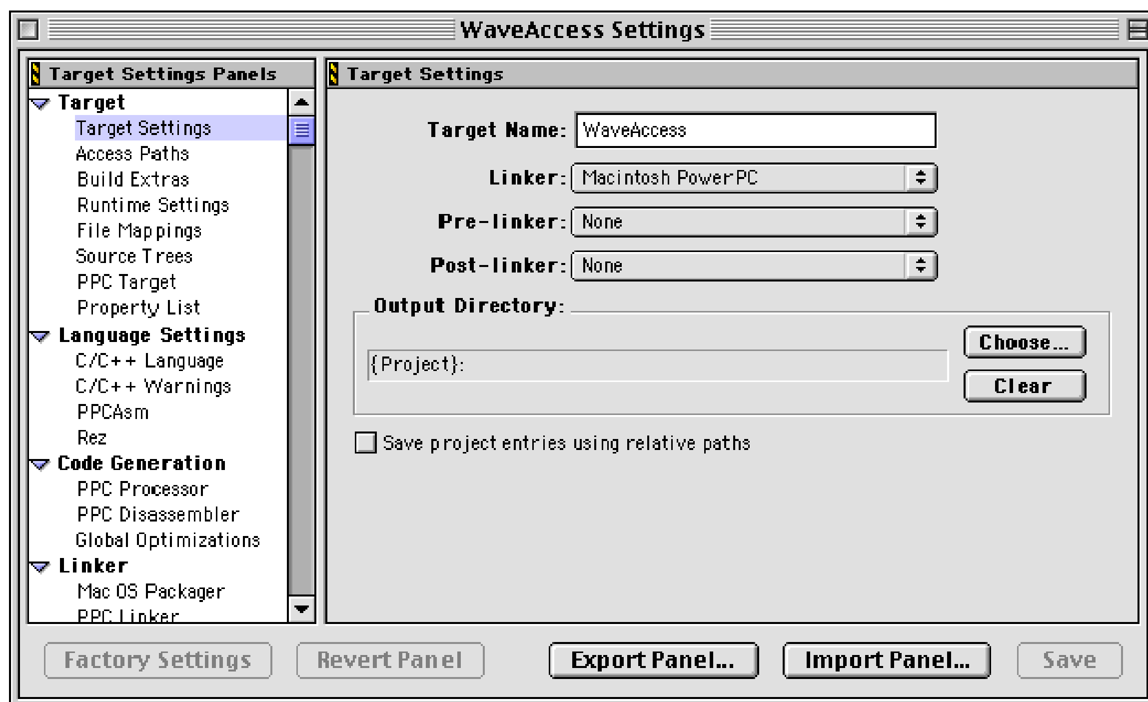
This screen shot of the project window shows the source files and libraries that are used in the project.




A CodeWarrior CFM XOP project is created from the CodeWarrior "C++ Toolbox Carbon" stationery and then configured as a "shared library" project. The following sections discuss the key project settings.

## Target Settings

When you create a new project, CodeWarrior creates two targets: a debug target and a final target. Each target has independent settings. To simplify matters we have created the sample CodeWarrior projects with just one target.

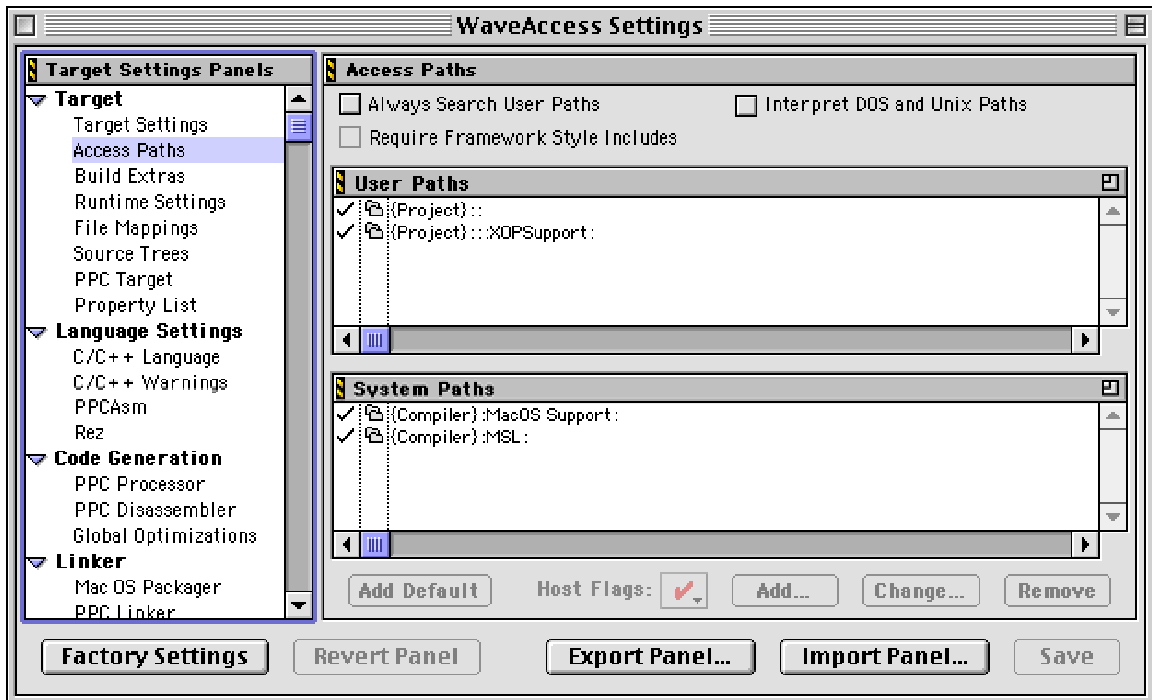


## Access Paths

The Access Paths tell CodeWarrior which directories to search for included header files. In the User Paths section "{Project}:" would refer to the folder containing the project file. Since our WaveAcces.h header file is up one level from that folder, we have added "{Project}::" which refers to the parent of the folder containing the project file, which is the WaveAccess directory.

{Project}:::XOPSupport: refers to the XOPSupport folder which is at the same level as the WaveAccess folder in the IgorXOPs5 directory. We need this access path because we #include XOPSupport header files.

We have created these access paths as project-relative (using the Add or Change buttons) so that they will continue to work if we move the IgorXOPs5 folder to another location on our hard disk or to another machine.




## Runtime Settings

In this pane you enter the path to your Igor Pro application file. When you debug your XOP, CodeWarrior will launch Igor Pro. Sometimes after entering the path you need to close and reopen the project or quit and restart CodeWarrior to make things work.

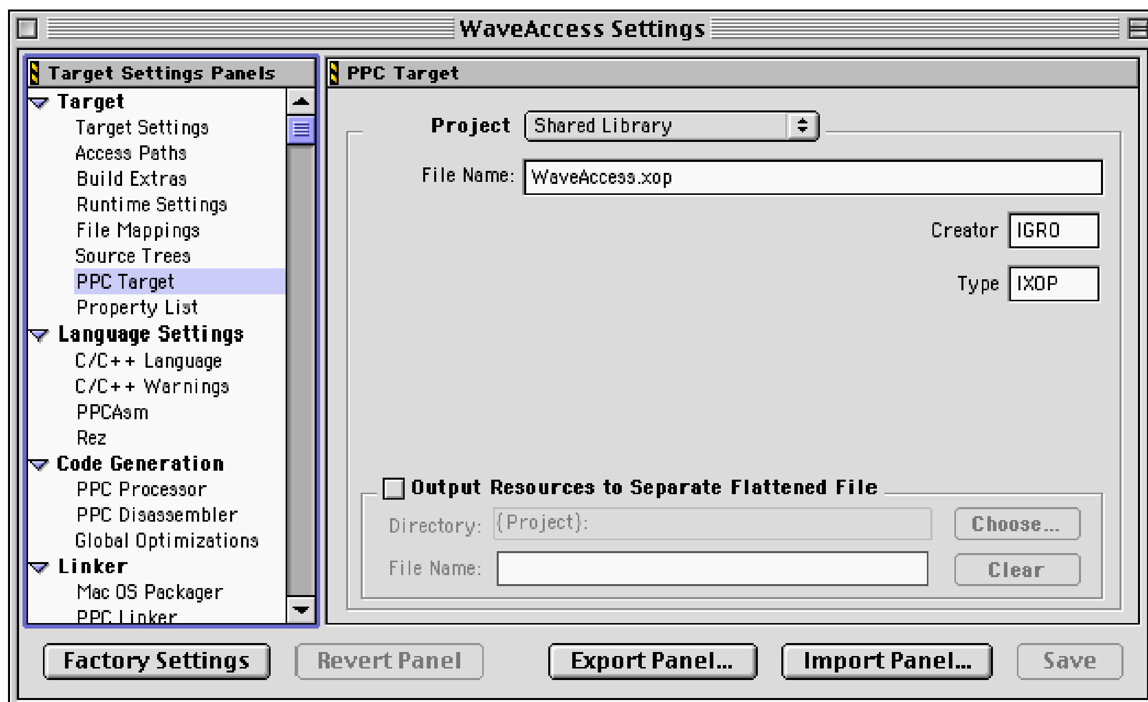
## PPC Target

Note that the project type is Shared Library.

By convention, the XOP file name ends with ".xop".

When Igor is launched, it scans the Igor Extensions folder and subfolders for files of type IXOP. If your CFM XOP file does not have this file type, Igor will not consider it to be an XOP.

The creator code is IGR0 (last character is zero).




## C/C++ Language

We specify MacHeadersCarbon.h as the prefix file. This is a precompiled version of headers including Apple's Carbon API headers and the Metrowerks' Standard C library (MSL) headers.

## PPC Processor

In the sample XOPs, we specify structure alignment as "68K", which means that fields are aligned on two-byte boundaries. This guarantees that structures passed between Igor and the XOP use the alignment required by the XOP Toolkit. See **Structure Alignment** on page 279 for details.

## Global Optimizations

For debugging, optimizations must be set to off. Otherwise the CodeWarrior debugger's behavior is erratic. Even with optimizations off, the CodeWarrior debugger sometimes shows incorrect values for variables.

When compiling a final version, you might want to set optimizations to Level 2. This might provide increased speed. However it also sometimes unmasks compiler bugs or your own bugs which were asymptomatic when compiling with optimizations off. So you should thoroughly retest after compiling with optimizations on.

## PPC Linker

The Main entry point must be set to main, the XOP's main function.

The Initialization and Termination entry points must be set to __initialize and __terminate. Failure to enter the initialization and termination entry points as shown can cause a crash when a C++ XOP throws an exception.

## Debugging a CodeWarrior CFM XOP

Here are the steps that you can take to debug a CFM XOP in CodeWarrior.

1. If you are running on Mac OS 9, check the Extensions folder of your System folder. It should contain a file called MetroNub, placed there by the CodeWarrior installer. If it is not there, find MetroNub on the CodeWarrior Mac OS Tools CD-ROM and drag it into your System:Extensions folder.
2. If you are running on Mac OS X, make sure that you are using CodeWarrior Pro 8.3 or later and that you have the latest Developer Tools from Apple. CodeWarrior uses the GDB debugger supplied with the developer tools. Prior to CodeWarrior 8.3 and OS X 10.3, there were compatibility problems between CodeWarrior and GDB.
3. Quit Igor if it is running.
4. In CodeWarrior Pro, open your XOP project file.
5. Using the Edit menu, open the target settings window (e.g., WaveAccess Settings). In the Runtime Settings pane, under the section Host Application, click the Choose button and find your Igor Pro application file. This is the application that CodeWarrior will launch when you choose Debug from the Project menu. If you have more than one copy of Igor Pro on your hard disk, make sure to choose the right one. Sometimes after entering the path you need to close and reopen the project or quit and restart CodeWarrior to make things work.
6. In the Global Optimizations pane, set the optimization to “Off”. If optimization is on, CodeWarrior will sometimes fail to stop at breakpoints or give bogus readouts of variables.
7. Click the Save button and then close the target settings window.
8. In the project window, in the "bug" column (the one under the green bug-like icon) click to add a bullet for each source file that you might want to debug. The bullet indicates that the compiler will generate debugging symbols for that file. You may as well turn debugging symbols on for all of your C or C++ source files.
9. In the Project menu, choose Make. When the project has finished compiling, go into the Finder and find the compiled XOP file. Make an alias for that file and drag the alias into the Igor Extensions folder in the folder containing your Igor Pro application. This is where Igor looks for XOPs when it is launched. (If you are running Igor Pro 4, see **The Igor Extensions Folder On Macintosh** on page 9).
10. Open one of your source files and find the point where you want to break. For example, if you want to break when your XOP is first launched by Igor, open your main source file (e.g., WaveAccess.c), find the main routine, and set a breakpoint somewhere in that routine. To set a breakpoint, click in the column that runs down the left edge of the source window, on line

of source code where you want to break. CodeWarrior will add a red bullet to indicate that a breakpoint has been set there.

11. Choose Project->Debug. CodeWarrior will launch the Igor application that you chose.
12. If the XOP adds a function to Igor, Igor will launch the XOP during Igor's initialization. If you set a breakpoint in your main routine, you should break into the debugger at this time. If the XOP adds command line operations but no functions, Igor will not launch the XOP during Igor's initialization.
13. If your breakpoint is not in your main function, do something that causes the source line where you set the breakpoint to execute. For example, choose your XOP's menu item, invoke its command line operation or invoke its function from the Igor command line.You should break into the debugger at this time.

## Mach-O XOP Projects in CodeWarrior Pro 8

The main reason to create a Mach-O binary XOP rather than a CFM XOP is if you need to use Mach-O frameworks, such as Apple's IOKit framework. You can create a Mach-O XOP using either Xcode or CodeWarrior. Mach-O XOPs run on Mac OS X only, not on Mac OS 9.

We expect that most CodeWarrior XOP development will use CFM, not Mach-O. For simplicity's sake the CodeWarrior sample XOPs are all CFM XOPs (except for the VDT2 XOP which uses IOKit). However, if you want to use CodeWarrior to create a Mach-O XOP, this section explains how.

Here is how you would create a CodeWarrior Mach-O XOP project for a project named MyXOP:

1. In CodeWarrior, choose File->New. This displays the New window.
2. Enter a project name (e.g., MyXOP.mcp) and set the project location (e.g., /IgorXOPs5/MyXOP/CW8).
3. Select Mac OS C++ Stationery from the list.
4. Click OK. This displays the New Project window.
5. Select "Mac OS X Mach-O"->"Mac OS Toolbox"->"C++ Toolbox Mach-O".
6. Click the OK button. This creates the project.
7. Remove the sample source files (SimpleAlert.c, SimpleAlert.rsrc, SimpleAlert.plc) from the project.
8. In the Finder, delete all files whose names start with SimpleAlert from the /IgorXOPs5/MyXOP/CW8 folder.
9. Copy the file /IgorXOPs5/VDT2/CW8/VDT2.plc to your project folder (IgorXOPs5/MyXOP/CW8). Rename this file as MyXOP.plc.
10. In CodeWarrior, open MyXOP.plc and change all of the references to VDT2 to references to MyXOP.

The project as created by CodeWarrior contains two targets, a debug target and a final target. We will discuss adjusting the settings for the debug target only. If you want to maintain two targets, you would need to make the same settings changes for the final target.

11. Open the project settings window and enter the following settings:

**Target**

***Target Settings***

*Target Name*: MyXOP Debug

*Linker*: Mac OS X PowerPC Mach-O

***Access Paths***

Click in the User Paths section to activate it.

Click the Add button and add the /IgorXOPs5/MyXOP folder as a project-relative access path. The resulting path should be displayed as:
`{Project}::`

Click the Add button and add the /IgorXOPs5/XOPSupport folder as a project-relative access path. The resulting path should be displayed as:
`{Project}:::XOPSupport`

Click the System Paths section to activate it.

Click the Add button and add

{Compiler}/MacOS Support/Universal/Interfaces/Rincludes

as a CodeWarrior-relative access path.

***Runtime Settings***

Click the Choose button and find your Igor Pro 5 application file.

***PPC Mac OS X Target***

*Project Type*: Bundle Package

*Bundle Name*: MyXOP.xop

*HFS Creator*: IGR0 (last character is zero)

*HFS Type*: BNDL

**Language Settings**

***C/C++ Language***

*Prefix File*: MSL MacHeadersMach-O.h

*C/C++ Warnings*: Uncheck all except Unused Variables and Extended Error Checking.

***Rez***

*Prefix File*: MyXOPPrefix.r (You will create this file later.)

**Code Generation**

***PPC CodeGen Mach-O***

*Struct Alignment*: 68K (See page 279 for details.).

**Linker**

***Mac OS Packager***

*Use Mac Packager*: Checked

*Create alias to Classic executable*: Unchecked

*Create PkgInfo file*: Checked

*Package Creator Type*: IGR0 (last character is zero)

*Package File Type*: IXOP

***PPC Mac OS X Linker***

*Export Symbols*: Use “.exp” file

*Main Entry Point*: _main

12. Save the project settings and close the project settings window.
13. In the project window, click the Sources icon and then choose Project->Add Files, and add the MyXOP.c, MyXOP.r and MyXOP.plc files to all targets.
14. Using Project->Add Files, add the console.stubs.c file to all targets. The file is in:

    /Metrowerks CodeWarrior/MSL/MSL_C/MSL_MacOS/Src/
15. Select the crt1.o (the C runtime library for an application) icon in the project window.
16. Using Project->Add Files, add the file bundle1.o (the C runtime library for a bundled package) to all targets. This should be in /usr/lib on your OS X volume.
17. Control-click the crt1.o icon and choose Clear to remove it from the project.
18. Using Project->Add Files, add to all targets the XOPSupport Mach.Lib file from /IgorXOPs5/XOPSupport/CW8.
19. Create a new text file named MyXOPPrefix.r and store it in the /IgorXOPs/MyXOP/CW8 folder. Enter the following text in the file:

    ```
    #define TARGET_RT_MAC_MACHO
    ```
20. Choose Project->Make.

    When you successfully compile and link for the first time, CodeWarrior will create a file named MyXOP.mcp.exp in the same folder as the CodeWarrior project.
21. Edit the MyXOP.mcp.exp file and remove all entries except for the entry for "_main". We now need to force CodeWarrior to relink.

22. Choose Project->Remove Object Code and remove object code from all targets.
23. Choose Project->Make to rebuild the XOP.
24. In the Finder, locate /MyXOP/CW8/MyXOP.xop. This is your compiled XOP package.
25. Make an alias from MyXOP.xop and put the alias in the Igor Extensions folder.

You are now ready to test your XOP.

## Debugging a CodeWarrior Mach-O XOP

Debugging a CodeWarrior Mach-O XOP works the same as debugging a CFM XOP. See **Debugging a CodeWarrior CFM XOP** on page 73.

# XOPs in Xcode

Xcode is Apple’s development system for Mac OS X. It is provided by Apple at no charge as part of their developer tools package. Xcode runs on Mac OS X 10.3 or later and generates Mach-O executables which can run on Mac OS X only.

As of this writing, Apple is shipping Xcode version 1.1. Since Xcode is a new development system, we recommend that you update to the current version of Xcode and Mac OS X before using it.

Igor Pro 4 does not support Mach-O XOPs, so you can use Xcode only for XOPs that will run with Igor Pro 5.00 or later. You also need XOP Toolkit 5 or later.

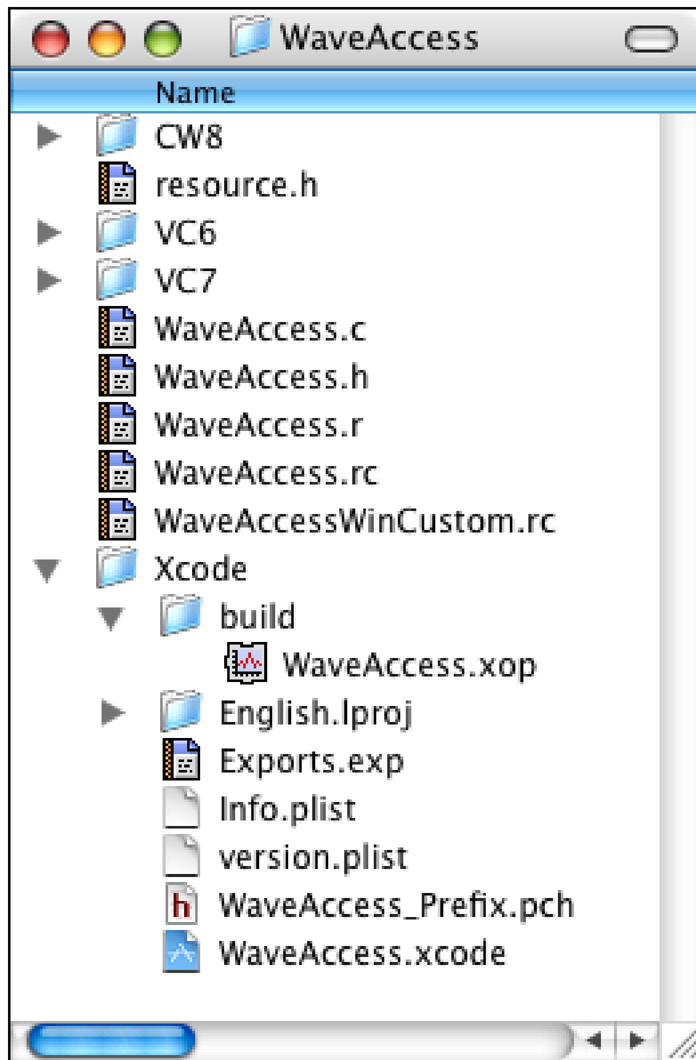
XOP Toolkit 5 provides sample projects and support files for Xcode 1.1. These samples and support files should work in later versions of Xcode, although minor tweaks may be necessary.

If you have not already done it, now would be a good time to read the Xcode help (choose Help->Xcode Help) so that you have an understanding of the basic Xcode concepts.

# XOP Projects in Xcode

This section provides the background information needed to understand how to create XOP projects in Xcode. For step-by-step instructions on creating a project, see **Creating a New Project** on page 35 and Creating the New Project In Xcode on page 41.

We will use the WaveAccess sample XOP as a case in point. The WaveAccess folder is inside the IgorXOPs5 folder and looks like this:



The Xcode project files are inside the Xcode folder, to keep them separate from the files for the other development systems. The discussion assumes this arrangement and we recommend that you use it. Note that the compiled XOP is in the WaveAccess/Xcode/build folder.

The build and English.lproj folders as well as the Info.plist, version.plist and WaveAccess_Prefix.pch files are created by Xcode. You may see other Xcode-created folders and files.

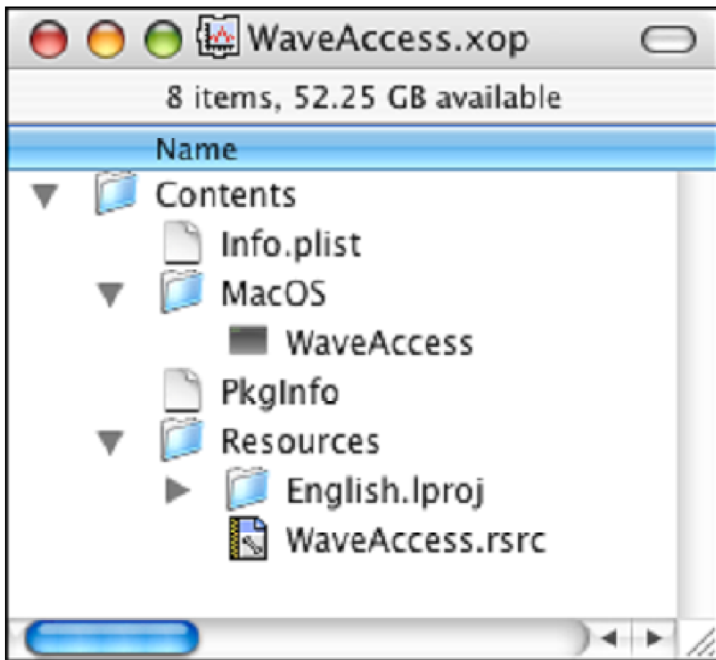
WaveAccess.xcode is the project file and contains the project settings. (Actually it is a package folder, not a file.) The build folder contains data created by the compiler such as object code as well as the compiled XOP.

WaveAccess.c contains the project source code while WaveAccess.h contains the project headers.

WaveAccess.r contains the XOP's Macintosh resources. The files resource.h, WaveAccess.rc and WaveAccessWinCustom.rc are used on Windows only.

The Exports.exp file is a plain text file that tells Xcode what routines the XOP needs to "export". In order for Igor to find a function in the XOP by name, it needs to be exported. The only function that needs to be exported for an XOP is main, so the Exports.exp file will be the same for all XOPs.

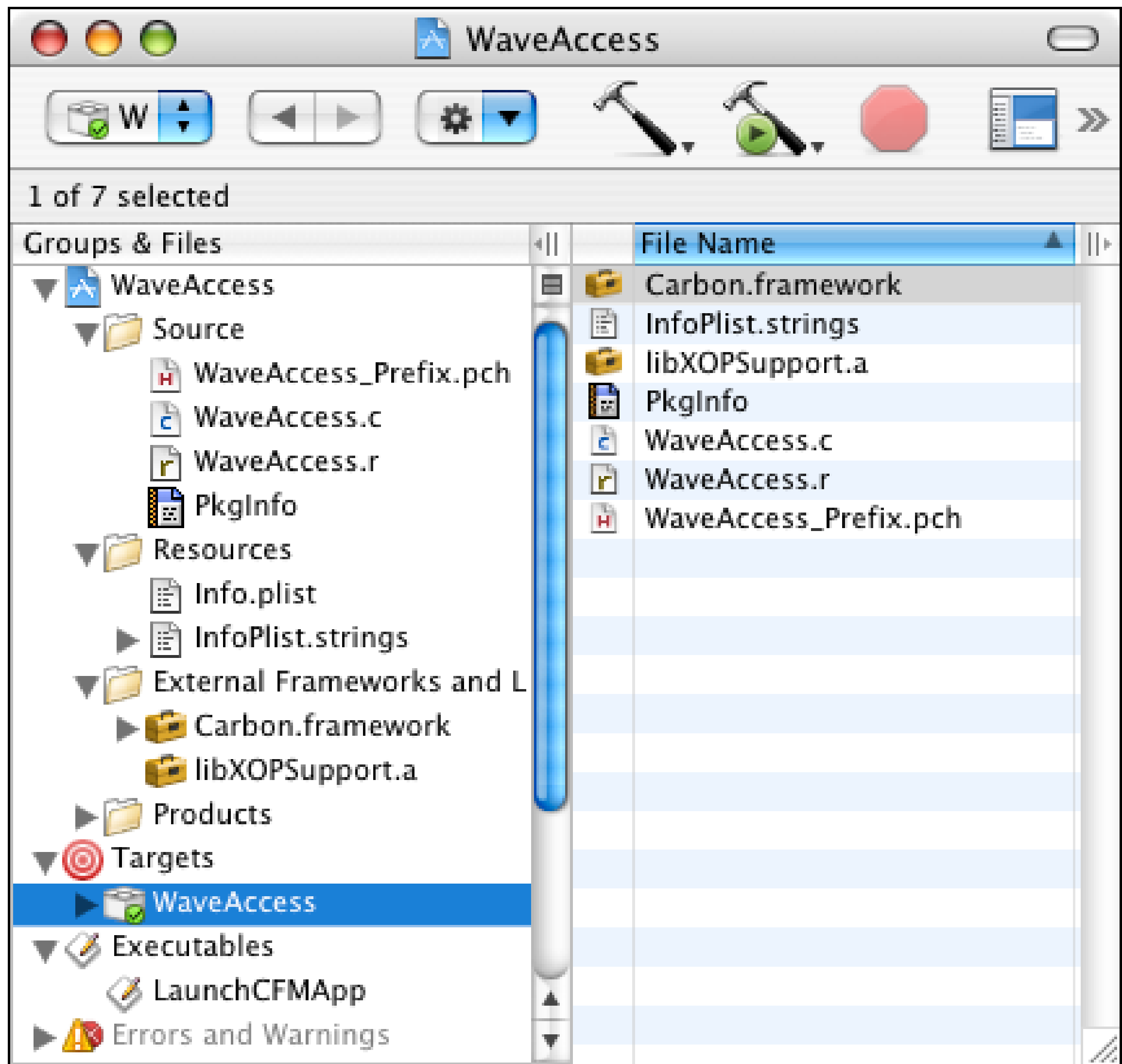
WaveAccess.xop appears to be a file but it is really a "package". If you control-click and and choose Show Package Contents, you see what is inside:



The layout of the package is as defined by Apple for a “packaged bundle”. Here “bundle” is Apple’s term for a plug-in.

The package is created automatically by Xcode when you compile the XOP. The actual executable code is in the file WaveAccess. The other files contain data describing the package and resources.

This screen shot of the project window shows the source files and libraries that are used in the project.



The WaveAccess_Prefix.pch file is automatically created by Xcode and used to speed up compilation by pre-compiling headers.

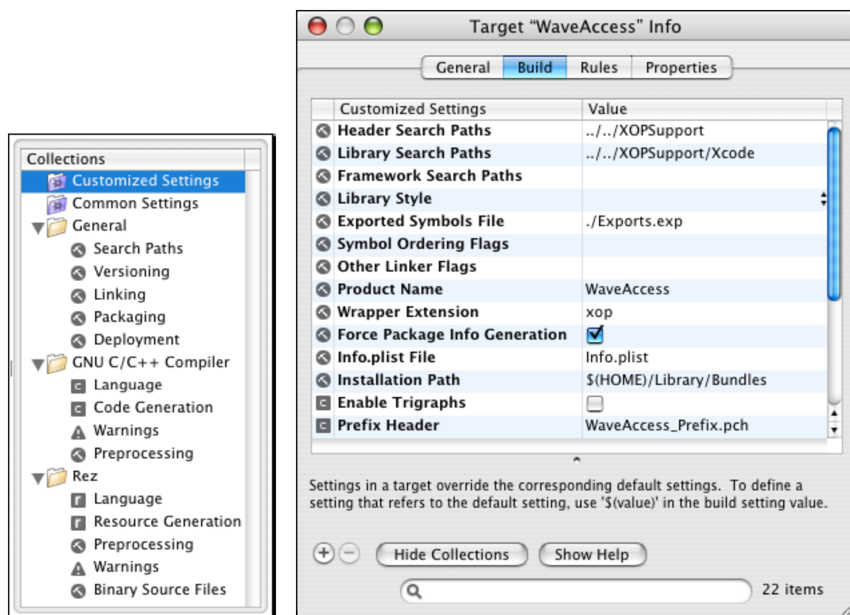
The info.plist and InfoPlist.strings files are automatically created by Xcode and used to store "meta-information" such as the XOP package's file type and creator codes.

The file Carbon.framework makes Apple's Carbon API accessible to the XOP. LibXOPSupport.a is the WaveMetrics XOPSupport library.

LaunchCFMApp is a system program that launches Code Fragment Manager applications. Igor Pro is a CFM binary executable. Xcode creates Mach-O binary executables. Xcode itself is not capable of launching CFM executables directly, so in order to debug the XOP, we need to use LaunchCFMApp to launch Igor Pro.

# Xcode Project Settings

If you select the Targets->WaveAccess icon in the Xcode Groups & Files pane and then choose Project->Get Info, and then click the Build tab, Xcode displays the project settings relating to compiling and linking the XOP. If you click the Customized Settings in the "Collections" tray, you see those settings which have non-default values.



The *Header Search Paths* setting allows #include statements to find the XOPSupport header files. The path is specified relative to the WaveAccess.xcode project file.

The *Library Search Paths* setting allows the Xcode version of the XOPSupport library to be found.

*Exported Symbols Files*: Specified as "./Exports.exp", meaning that Xcode is to look for the file Exports.exp in the project folder. Exports.exp declares the main function to be exported (i.e., visible to Igor Pro by name). This file will work for any XOP so if you create a new Xcode XOP, you can just duplicate the Exports.exp file from an existing Xcode XOP.

*Wrapper Extension*: Specified as ".xop". Igor identifies Mach-O XOP packages by the .xop extension on the package folder's name.

*Force Package Info Generation*: Checked. We're not really sure what this does and Apple's documentation is of little help.

*Prefix Header*: This is automatically set by Xcode when the project is created to "WaveAccess_Prefix.pch", a file which Xcode automatically creates. The file determines which header files are pre-compiled and stored in binary form with the project data to speed up compilation.

*OTHER_REZFLAGS*: Set to "-i ../../XOPSupport  -d TARGET_RT_MAC_MACHO". This specifies flags to pass to the Rez compiler when the .r file is compiled. The -i part allows #include statements to find the XOPResources.h file in the XOPSupport folder. The -d part predefines the symbol TARGET_RT_MAC_MACHO. This symbol is used in an ifdef in the XOPResources.r file to set a value (DEV_SYS_CODE) used in the XOPI resource in WaveAccess.r. Igor inspects the XOPI resource at launch time to see if the XOP is a Mach-O binary. Without this setting, the XOP will not work. Note that "MACHO" ends with the letter O, not the number zero.

If you click the Properties tab in the Target Info window, Xcode displays the project properties, including the project type (IXOP), project creator (IGR0) and version. These values are stored by Xcode in the Info.plist file which winds up in the XOP package.

The project type of IXOP sets the type of the XOP package as specified in Info.plist. It does not set the file type of the executable file, WaveAccess, which is inside the XOP package. It is important that the executable file's type *not* be set to IXOP as this would cause Igor to think that it was a CFM XOP.

# Xcode C++ Projects

If your XOP's main file is a C++ file (.cp or .cpp) rather than a C file (.c), you need to make a change. When compiling C++, Xcode requires that the main function return an int. Open your main .cpp file and change the main function from:

```
HOST_IMPORT void
main(IORecHandle ioRecHandle)
```

to:

```
HOST_IMPORT int
main(IORecHandle ioRecHandle)
```

Find the declaration of your main function and change it also. The declaration will be either in your main .cpp file or in a .h file such.

# Xcode XOP Package

The XOP package folder should appear in the Finder as if it were an XOP file and you should have to control-click it and choose Show Package Contents to see what the package folder contains. However, after compiling in Xcode, the XOP package incorrectly appears as a folder. Apple's documentation on this is unclear and figuring out how to make it work has been complicated by changes in the behavior of the Finder from one version of Mac OS X to another. Sometimes a restart is required to get the Finder to notice changes to the package.

We found a technique to work around the problem. It consists of putting a file named PkgInfo into the package. PkgInfo tells the Finder that the folder is a package and specifies the file type which the Finder uses to determine which icon to use to represent the package. This technique is obsolete, according to Apple's documentation, but we have found no other way to get it to reliably work.

In order to automatically put the PkgInfo file in the package, we added a "New Copy Files Build Phase" to the Targets->WaveAccess->Copy Files icon. It makes a copy of the PkgInfo file in the XOPSupport folder and puts the copy in the output package folder. Here are the steps we used:

1. In the Groups & Files list, select the WaveAccess->Source icon.
2. Choose Project->Add Files and add /IgorXOPs5/XOPSupport/PkgInfo. In the resulting dialog, choose Relative to Project from the Reference Type popup menu and click Add.
3. Select the Targets->WaveAccess icon.
4. Choose Project->New Build Phase->New Copy Files Build Phase. In the resulting Copy Files Info window, select Wrapper from the Destination popup menu, enter Contents in the Path edit box, and then close the window.
5. Drag the WaveAccess->Source->PkgInfo icon into the Targets->WaveAccess->Copy Files icon.
6. Choose Build->Build to rebuild the target.
7. In the Finder, close and reopen the /WaveAccess/Xcode/build folder to force a Finder refresh.
8. Now the WaveAccess.xop folder should appear like a file with an XOP icon. However, you may need to restart your computer to get the Finder to use the correct icon.

# Debugging an Xcode XOP

Some additional setup is required to debug your XOP. You need to tell Xcode what program hosts your XOP. And because Igor Pro is a CFM application which can not be executed directly from Xcode, the setup is a bit complicated. You will have to tell Xcode to launch the system's LaunchCFMApp program and then tell LaunchCFMApp to launch Igor Pro.

Here's how you would do after opening the WaveAccess sample project in Xcode:

1. First you need to determine the path to your LaunchCFMApp file. On our system it is located at:

   /System/Library/Frameworks/Carbon.framework/Versions/A/Support/LaunchCFMApp

   This will probably work for you too. If not, you can search for LaunchCFMApp in the Finder.
2. In the Xcode Groups & Files list, select the WaveAccess icon.
3. Choose Project->New Custom Executable.
4. Enter LaunchCFMApp as the Executable Name.

   Click the Choose Button and find your LaunchCFMApp file.

   Click Finish.
5. In Xcode 1.1 there is a bug which causes the icon added to the Executables group to be named “Executable” instead of LaunchCFMApp. To fix this, control-click the Executables->Executable icon, choose Rename, and change the name to LaunchCFMApp.
6. In the Groups & Files list, double-click Executables->LaunchCFMApp.

   Under Launch Arguments, click the + icon and enter the full path to your Igor Pro application in double-quotes (e.g., "/Applications/WaveMetrics/Igor Pro Folder/Igor Pro").

   Close the Executable window.
7. Choose Build->Build.

   When the project compilation is finished, go into the Finder and find the compiled XOP (/WaveAccess/build/WaveAccess.xop).
8. Make an alias for that XOP and drag the alias into the Igor Extensions folder in the folder containing your Igor Pro application. This is where Igor looks for XOPs when it is launched. (If you are running Igor Pro 4, see **The Igor Extensions Folder On Macintosh** on page 9).
9. In Xcode, open the WaveAccess->Source icon and select your main source file (e.g., WaveAccess.c) and set a breakpoint at the start of the main function. Click in the lefthand gutter to set the breakpoint.
10. Choose Debug->Debug Executable. Xcode will launch the LaunchCFMApp program which will launch your Igor Pro program.

11. If your XOP adds one or more external functions, Igor will launch it at this time, in which case you should break into Xcode at the point in the main function where you set the breakpoint. You may have to click the Xcode debug window to activate it.
12. If your XOP does not add external functions, do something to cause Igor to launch it, such as invoking your external operation from Igor's command line. You should break into Xcode at this point. You may have to click the Xcode debug window to activate it.

We have found that sometimes when you try to quit Igor after debugging in Xcode, Igor does not quit and you have to go into Xcode and chose Debug->Stop Debugging.

# Other Xcode Notes

## fopen Function

In Xcode, the fopen function in the standard file package expects to receive a POSIX (i.e., Unix) path. In CodeWarrior, fopen expects to receive an HFS (i.e., Mac OS) path. The XOPSupport XOPOpenFile function, which calls fopen, takes care of this problem. However, if you have used fopen in your own function under CodeWarrior, you will need to change your function to pass a POSIX path.

## Help Files

An XOP's help file must be placed in the same folder as the XOP itself. In Igor Pro 5.00, there was a bug which prevented Igor from finding an Xcode XOP's help file. This prevented Igor from finding XOP help for display in Igor's Help Browser, among other problems. The bug was fixed in Igor Pro 5.01. Since we are always fixing bugs, it is a good idea to update to the latest version of Igor Pro.

## Balloon Help

Xcode does not support balloon help resources. If your .r file includes Balloons.r you must remove the #include statement. If your .r file defines 'hmnu' or 'hdlg' resources, you must remove them.

## Structures Defined in Parameter Lists

Xcode does not like structures that are defined in parameter lists, like this:

```
static int
WAGetWaveInfo(
struct {
   waveHndl w;
   Handle strH;
}* p)
```

So we changed the XOP samples to define the structure separately, like this:

```
struct WAGetWaveInfoParams {
   waveHndl w;
   Handle strH;
};
typedef struct WAGetWaveInfoParams WAGetWaveInfoParams;

static int
WAGetWaveInfo(WAGetWaveInfoParams* p)
```

# XOPs in Visual C++ 6

Visual C++ 6, published by Microsoft, Inc., is a C and C++ development system that can build programs for Microsoft Windows. Visual C++ 6 runs on Windows and produces Windows XOPs.

Visual C++ 6 comes in standard (cheap), professional (expensive) and enterprise (very expensive) editions. The standard edition is fine for XOP development.

Microsoft is now shipping Visual C++ 7 (called Visual C++ .NET). If you go to buy Visual C++, you will no doubt find Visual C++ .NET. However, Visual C++ 6 is still supported for XOP development.

XOP programming does not require MFC (Microsoft Foundation Class). However, the Visual C++ 6 resource editor creates resource script files that refer to an MFC header file. Therefore, if you did not install MFC when you installed Visual C++ 6, you need to do the following steps.

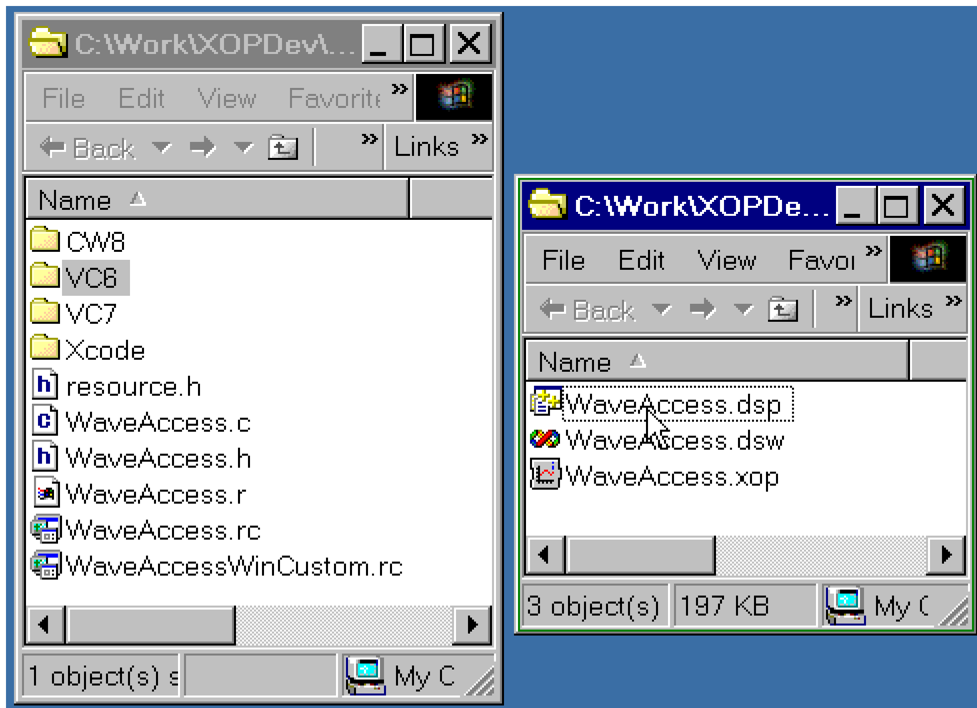
1. Find your "VC98" folder. This is usually located at:

   C:\Program Files\Microsoft Visual Studio\VC98
2. If your VC98 folder already contains an MFC folder, then you should not need the remaining steps.
3. Create a new folder named "MFC" in your VC98 folder.
4. Create a new folder named "Include" inside the MFC folder.
5. Find the file "afxres.h" from on your Visual C++ 6 CD-ROM and copy it to the newly created VC98\MFC\Include folder.

# XOP Projects in Visual C++ 6

This section provides the background information needed to understand how to create XOP projects in Visual C++ 6. For step-by-step instructions on creating a project, see **Creating a New Project** on page 35 and **Creating the New Project In Visual C++ 6** on page 45.

We will use the WaveAccess sample XOP as a case in point. The WaveAccess folder is inside the IgorXOPs5 folder and looks like this:




The Visual C++ 6 project files are inside the VC6 folder, to keep them separate from the files for the other development systems. The discussion assumes this arrangement and we recommend that you use it. Note that the compiled XOP is in the WaveAccess/VC6 folder.

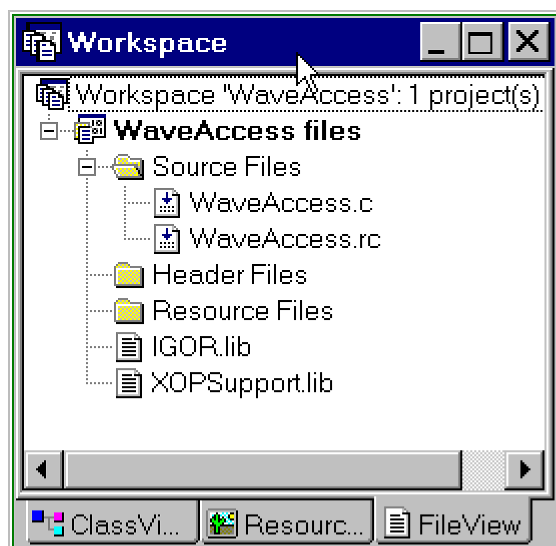
WaveAccess.dsp is the Visual C++ 6 project file and WaveAccess.dsw is the Visual C++ 6 workspace file. To open the project, double-click the workspace file. You may see other Visual C++ 6-created folders and files. They contain data created by Visual C++ 6, such as compiled object code. Only the .dsp and .dsw files are essential for recreating the project.

WaveAccess.c contains the project source code while WaveAccess.h contains the project headers.

WaveAccess.rc is the main resource file. It #includes resource.h (created by Visual C++), and WaveAccessWinCustom.rc (created by the XOP programmer). WaveAccess.rc contains standard Windows resources that are editable using the Visual C++ resource editor. WaveAccessWinCustom.rc contains XOP-specific resources that you edit as text. Igor examines these custom resources to determine what operations, functions and menus the XOP adds, among other things.

WaveAccess.r contains the Macintosh resources and is not used on Windows.

This screen shot of the workspace window shows the source files and libraries that are used in the project.




The project is set up to compile either a debug version or a release version of the XOP. You choose which will be built by choosing Build->SetActive Configuration. Both of these configurations create an XOP with the same name (e.g., WaveAccess.xop). During development, you will normally compile using the debug configuration. Once your XOP is in final form, you may want to try the release configuration to see if it runs significantly faster. The downside of switching to the release configuration is that switching sometimes unmasks programmer or compiler bugs that you might not catch. So you should thoroughly retest after compiling the release configuration.

# Visual C++ 6 Project Settings

If you select the Project->Settings, Visual C++ 6 displays the project settings dialog. Here is a discussion of the significant settings which need to be changed from the default. If you change a setting, in most cases you should change it for both the debug and release configurations.

### Debug Tab/General Category

*Executable for debug session*: Enter the path to your Igor.exe application file. Visual C++ will launch Igor when to debug your XOP.

### C/C++ Tab/Code Generation Category

*Use runtime-library*: Single-Threaded. We have not tried using the multi-threaded library and we don't know its ramifications.

*Struct member alignment*: 2 Bytes. This guarantees that structures passed between Igor and the XOP use the alignment required by the XOP Toolkit. See **Structure Alignment** on page 279 for details.

### C/C++ Tab/Preprocessor Category

*Additional include directories*: ..\..\XOPSupport. Allows #include statements to find the XOPSupport header files. The path is specified relative to the WaveAccess.dsp project file.

### Link Tab/General Category

*Output file name*: WaveAccess.xop. The normal Visual C++ arrangement is to have the output file created in the Debug or Release folder created by Visual C++ inside the project folder (WaveAccess\VC6). However we set up both the debug and release configurations to create the output file in the project folder so that a shortcut placed in the Igor extensions folder will always refer to the last compiled version.

*Object/library modules*: version.lib must be added to the default list of libraries.

### Link Tab/Input Category

*Ignore libraries*: libcd.lib.

### Resources Tab

*Additional include directories*: ..\..\XOPSupport. Allows #include statements to find the XOPResources.h file. The path is specified relative to the WaveAccess.dsp project file.

## Debugging a Visual C++ 6 XOP

Here are the steps that you can take to debug a Visual C++ 6 XOP.

1. Open the workspace (e.g., WaveAccess.dsw) in Visual C++ 6.
2. Use Build->Set Active Configuration to activate the debug configuration of the project.
3. Build the XOP using the “Build <XOP Name>” item in the Build menu.
4. On the desktop, create a shortcut from the executable XOP file (e.g., WaveAccess\VC6\WaveAccess.xop) that you just built and put the shortcut in the Igor Extensions folder.
5. Back in Visual C++ 6, open your main source file (e.g., WaveAccess.c), and set a breakpoint at the start of the main function. Press F9 to set a breakpoint.
6. Press F5. This will launch Igor Pro.
7. If your XOP adds one or more external functions, Igor will launch it at this time, in which case you should break into Visual C++ at the point in the main function where you set the breakpoint.
8. If your XOP does not add external functions, do something to cause Igor to launch it, such as invoking your external operation from Igor's command line. You should break into Visual C++ at this point.

# XOPs in Visual C++ 7 (.NET)

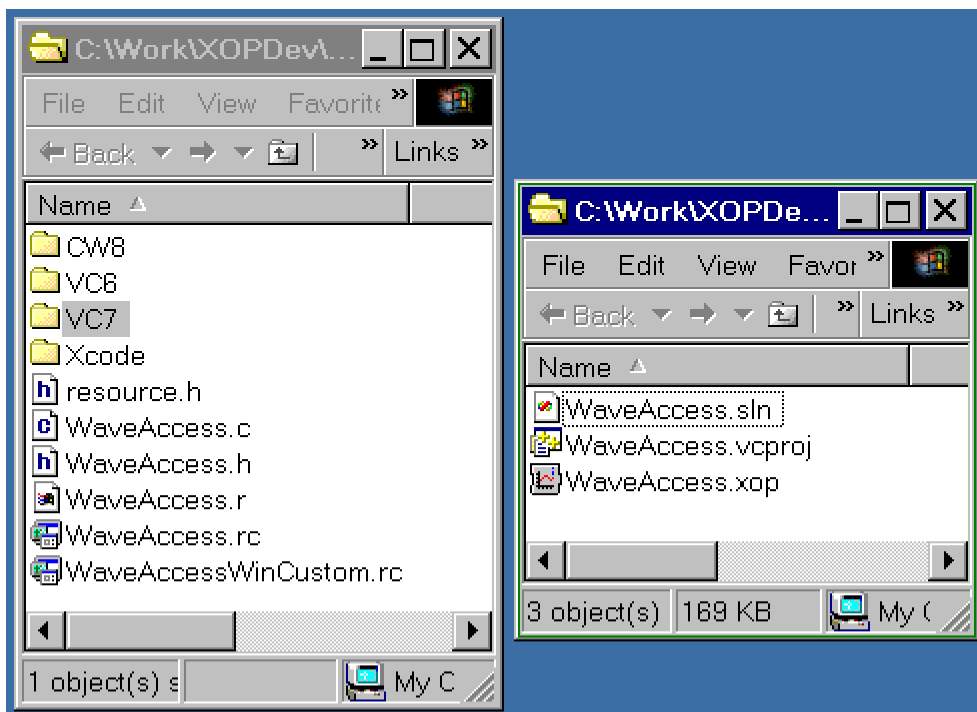
Visual C++ 7, published by Microsoft, Inc., is a C and C++ development system that can build programs for Microsoft Windows. It is better known as Visual C++ .NET. Visual C++ 7 runs on Windows and produces Windows XOPs.

Visual C++ 7 comes in standard (cheap), professional (expensive) and enterprise (very expensive) editions. The standard edition is fine for XOP development.

# XOP Projects in Visual C++ 7

This section provides the background information needed to understand how to create XOP projects in Visual C++ 7. For step-by-step instructions on creating a project, see **Creating a New Project** on page 35 and **Creating the New Project In Visual C++ 7 (.NET)** on page 49.

We will use the WaveAccess sample XOP as a case in point. The WaveAccess folder is inside the IgorXOPs5 folder and looks like this:




The Visual C++ 7 project files are inside the VC7 folder, to keep them separate from the files for the other development systems. The discussion assumes this arrangement and we recommend that you use it. Note that the compiled XOP is in the WaveAccess/VC7 folder.

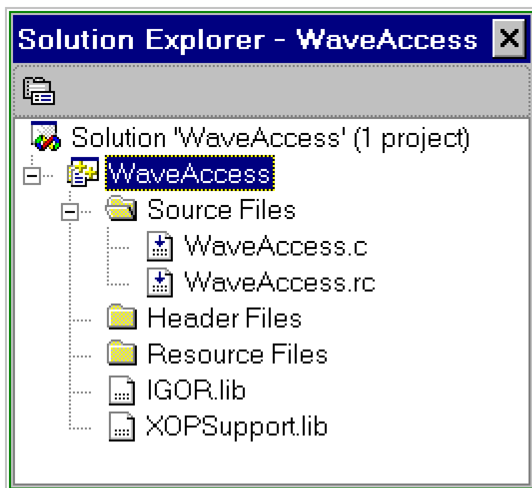
WaveAccess.vcproj is the Visual C++ 7 project file and WaveAccess.sln is the Visual C++ 7 “solution” file. To open the project, double-click the solution file. You may see other Visual C++ 7-created folders and files. They contain data created by Visual C++ 7, such as compiled object code. Only the .vcproj and .sln files are essential for recreating the project.

WaveAccess.c contains the project source code while WaveAccess.h contains the project headers.

WaveAccess.rc is the main resource file. It #includes resource.h (created by Visual C++), and WaveAccessWinCustom.rc (created by the XOP programmer). WaveAccess.rc contains standard Windows resources that are editable using the Visual C++ resource editor. WaveAccessWinCustom.rc contains XOP-specific resources that you edit as text. Igor examines these custom resources to determine what operations, functions and menus the XOP adds, among other things.

WaveAccess.r contains the Macintosh resources and is not used on Windows.

This screen shot of the Solution Explorer window shows the source files and libraries that are used in the project.




The project is set up to compile either a debug version or a release version of the XOP. You choose which will be built choosing Build-> Configuration Manager. Both of these configurations create an XOP with the same name (e.g., WaveAccess.xop). During development, you will normally compile using the debug configuration. Once your XOP is in final form, you may want to try the release configuration to see if it runs significantly faster.

# Visual C++ 7 Project Settings

If you choose View->Solution Explorer and then right-click the WaveAccess icon and then choose Properties, Visual C++ 7 displays the project settings. Here is a discussion of the significant settings which need to be changed from the default. If you change a setting, in most cases you should change it for both the debug and release configurations.

### Debugging Properties

*Command*: Enter the path to your Igor.exe application file. Visual C++ will launch Igor when to debug your XOP.

### C/C++ Properties /General Category

*Additional include directories*: ..\..\XOPSupport. Allows #include statements to find the XOPSupport header files. The path is specified relative to the WaveAccess.vcproj project file.

### C/C++ Properties /Code Generation Category

*Runtime Library*: Single-Threaded. We have not tried using the multi-threaded library and we don't know its ramifications.

*Struct member alignment*: 2 Bytes. This guarantees that structures passed between Igor and the XOP use the alignment required by the XOP Toolkit. See **Structure Alignment** on page 279 for details.

### Linker Properties /General Category

*Output file name*: WaveAccess.xop. The normal Visual C++ arrangement is to have the output file created in the Debug or Release folder created by Visual C++ inside the project folder (WaveAccess\VC7). However we set up both the Debug and Release configurations to create the output file in the project folder so that a shortcut placed in the Igor extensions folder will always refer to the last compiled version.

### Linker Properties /Input Category

*Additional Dependencies*: version.lib must be added to the default list of libraries.

*Ignore libraries*: libcd.lib.

### Resources Properties

*Additional include directories*: ..\..\XOPSupport. Allows #include statements to find the XOPResources.h file. The path is specified relative to the WaveAccess.vcproj project file.

# XOPSupport Warnings

When you build an XOP in Visual C++ 7, you may see LNK4204 warnings saying that XOPSupport.lib is missing debugging information. This is of no consequence unless you want to step into XOPSupport routines while debugging. In that case, you can fix the problem by opening IgorXOPs5\XOPSupport\VC7\XOPSupport.sln in Visual C++ 7 and rebuilding the XOPSupport library.

# Debugging a Visual C++ 7 XOP

Here are the steps that you can take to debug a Visual C++ 7 XOP.

1. Open the solution (e.g., WaveAccess.sln) in Visual C++ 7.
2. Use Build-> Configuration Manager to activate the debug configuration of the project.
3. Build the XOP using the Build <XOP Name> item in the Build menu.
4. On the desktop, create a shortcut from the executable XOP file (e.g., WaveAccess\VC7\WaveAccess.xop) that you just built and put the shortcut in the Igor Extensions folder.
5. Back in Visual C++ 7, open your main source file (e.g., WaveAccess.c), and set a breakpoint at the start of the main function. Press F9 to set a breakpoint.
6. Press F5. This will launch Igor Pro.
7. If your XOP adds one or more external functions, Igor will launch it at this time, in which case you should break into Visual C++ at the point in the main function where you set the breakpoint.
8. If your XOP does not add external functions, do something to cause Igor to launch it, such as invoking your external operation from Igor's command line. You should break into Visual C++ at this point.

# Writing XOPs in C++

Writing an XOP in C++ is not significantly different from writing it in C. Depending on the platform that you are writing for and your development system, you may need to perform a few additional steps, as we outline below.

## Mixing C and C++ Code

In general, your C++ XOP code will use XOPSupport routines or other functions that obey the C calling convention. To use a C routine in your C++ code, you must use the extern "C" declaration when declaring C routine. For example, if the C function foo is declared in its original C file as:

```
void foo(int i);
```

you need to add to any C++ file that contains a call to the function the following declaration:

```
extern "C" void foo(int i);
```

To simplify the process, header files for most libraries, including the XOPSupport headers, have the following structure:

```
#ifdef __cplusplus
extern "C" {
#endif

...

file definitions, declarations and prototypes

...

#ifdef __cplusplus
}
#endif
```

This structure ensures that when the header file is included in a C++ module, it automatically contains proper extern declarations for all of its C functions. Because the XOPSupport headers have this structure, you don't need to worry about extern declarations when you use XOPSupport routines in your C++ files.

## Code Changes For C++

A regular .c source file may compile without errors using the C compiler. If you try to compile the file again after saving it with the .cpp suffix and changing the compiler settings to activate the C++ compiler, you may get several error messages. These errors are typically associated with stricter type checking rules under C++. For example,

```
float *fp= WaveData(p->waveHandle);
```

will generate an error: illegal implicit conversion from 'void *' to 'float *'. To eliminate the error you should make the explicit conversion:

```
float *fp= (float *)WaveData(p->waveHandle);
```

## Using C++ Exceptions

You can use standard C++ exceptions in your XOPs. To do so, you will have to set the appropriate items in your project settings. These are discussed in the following sections.

In implementing C++ exceptions, it is very important that your final catch block always be before code execution returns to Igor. There are no catch blocks in Igor. Exceptions thrown from within your XOP must be caught inside your XOP. Also, depending on the objects that you choose to throw, you must make sure that you delete these objects before execution returns to Igor.

When writing a complex XOP it is sometimes useful to employ a catch-all wrapper. In the example below we show how this is done for an operation:

```
static void
XOPEntry(void)
{
   long result=0;

   try {
      switch(GetXOPMessage()) {
         case CMD:
            result = DoCmd();
            break;
      }
   }

   catch(...) {   // This will catch any exception.
      result = someErrorCode;
   }
   SetXOPResult(result);
}
```

The catch-all wrapper guarantees that all exceptions are caught. However, because it is so general, it usually does not contain provisions for disposing of thrown objects.

# C++ XOPs in CodeWarrior Pro

## CodeWarrior CFM Project Settings

In Linker pane of the project settings window, you must set the entry points correctly.

The Main entry point must be set to main, the XOP's main function.

For C++ XOPs, the Initialization and Termination entry points must be set to __initialize and __terminate. Failure to enter the initialization and termination entry points as shown can cause a crash when a C++ XOP throws an exception.

## Using the new Operator in CodeWarrior

C++ objects are usually allocated using the new operator. For example, an array of integers may be allocated by:

```
int *myArray;

myArray = new int[40]
if (myArray == NULL)
   DoErrorHandling();
```

CodeWarrior's implementation of the new operator depends on the definition of NEWMODE in the file New.cp (in the "MacOS Support:Libraries:RunTime:(Common Sources)" folder). The standard libraries shipped by Metrowerks use the "fast" mode which allocates memory in blocks that are not released even after the corresponding objects are destroyed. If your XOP is transient (that is, if you allow Igor to remove it from memory when your XOP is finished doing its job) this gives rise to severe memory leaks. If your XOP is resident (stays in memory until Igor quits), then this is not a problem.

For transient XOPs, the solution to this problem is to redefine NEWMODE at the top of the file New.cp

```
   #define NEWMODE NEWMODE_SIMPLE
```

You then need to recompile all the CodeWarrior runtime libraries that you use. This causes CodeWarrior to generate code that uses different memory management techniques, and avoids the leakage problem.

To avoid the need to recompile Metrowerks libararies, we recommend that you leave your XOP resident (the default) and not bother with changing the behavior of new. Resident and transient XOPs are discussed on page 135.

# C++ XOPs in Xcode

If your XOP's main file is a C++ file (.cp or .cpp) rather than a C file (.c), you need to make a change. When compiling C++, Xcode requires that the main function return an int. Open your main .cpp file and change the main function from:

```
HOST_IMPORT void
main(IORecHandle ioRecHandle)
```

to:

```
HOST_IMPORT int
main(IORecHandle ioRecHandle)
```

Find the declaration of your main function and change it also. The declaration will be either in your main .cpp file or in a .h file such.

# C++ XOPs in Visual C++ 6

If you use C++ exceptions, you need to enable the exception handling in the C/C++ tab, C++ Language category, in the project settings dialog.

If you use RTTI you need to enable it in the C/C++ tab, C++ Language category, in the project settings dialog.

# C++ XOPs in Visual C++ 7

The default settings in a new project should work with possible minor tweaks.

*Chapter* 4

# Igor/XOP Interactions


Overview ........ 105
XOP Resources ........ 108
Creating Resources on Macintosh ........ 109
Creating Resources on Windows ........ 109
The XOPI 1100 Resource ........ 110
Basic XOP Messages ........ 112
Messages for Operations and Functions ........ 115
Messages for XOPs with Windows ........ 116
Messages for XOPs that Save and Load Settings ........ 124
XOP Errors ........ 127
Error Codes ........ 127
Igor Error Codes ........ 127
XOP Custom Error Codes ........ 128
Mac OS Error Codes ........ 128
Handling Windows OS Error Codes ........ 128
Adding Custom Errors ........ 130
Displaying Your Own Error Alert ........ 131
XOPs and Preferences ........ 132
XOPs and Experiments ........ 132
Saving and Loading XOP Settings ........ 133
The IORecHandle ........ 135
Resident and Transient XOPs ........ 135
Receiving IDLE Messages For Background Processing ........ 136
Messages, Arguments and Results ........ 136
Handling Recursion ........ 137
Data Sharing ........ 139
Igor/XOP Compatibility Issues ........ 140
Checking Igor's Version ........ 140

XOP Protocol Version ........................................................................ 141

# Overview

The code on page 106 is a simplified sketch XOP1, a simple XOP that adds a single command line operation to Igor. The underlined items are functions, constants or structures defined in the XOPSupport library and headers. To get a feeling for how Igor and the XOP interact, we will follow a chain of events in chronological order.

1. When Igor is launched, it scans the Igor Extensions folder and any subfolders looking for XOPs. On Macintosh, Igor identifies a CFM XOP file by its file type (IXOP) and identifies a Mach-O XOP package by the ".xop" extension on the package folder's name. On Windows, Igor identifies an XOP file by its ".xop" file name extension. When Igor finds an XOP, it first looks for its XOPI resource, from which Igor determines various characteristics of the XOP.
2. Next Igor looks for resources that identify what things (external operations, external functions, menu items, menus) the XOP adds. In the example below, the XOPC resource tells Igor that the XOP adds a command line operation named XOP1. This resource is stored in the XOP's resource fork and is defined in the XOP1.r file on Macintosh and in the XOP1WinCustom.rc file on Windows.
3. The user invokes the XOP1 operation via Igor's command line or via a procedure.
4. Igor loads the XOP into memory and calls the XOP's main routine, passing it an IORecHandle. main does all of the XOP's initialization including calling the following XOPSupport routines.

| | |
|---|---|
| XOPInit | Stores the IORecHandle in a global variable which is used by all subsequent XOPSupport calls. |
| SetXOPEntry | Sets a field in the IORecHandle pointing to the XOP's XOPEntry routine. Igor uses this routine for most subsequent calls to the XOP. |
| RegisterOperation | Tells Igor the syntax of the XOP1 external operation and the address of the function (ExecuteXOP1) that Igor must call called to execute the operation. |
| SetXOPResult | Sets the result field in the IORecHandle to zero. This tells Igor that the initialization succeeded. |

5. Igor parses the command's parameters, stores them in a structure, and calls ExecuteXOP1, passing the structure to it.
6. The ExecuteXOP1 function does the necessary work and returns a result to Igor. The result is zero for success or a non-zero error code..
7. If the result returned to Igor is non-zero, Igor displays an error alert.

```
// From XOP1.r for Macintosh.
resource 'XOPC' (1100) {   // Describes operations the XOP adds to Igor.
    {
        "XOP1",                             // Name of operation.
        XOPOp + UtilOP + compilableOp,      // Operation's category.
    }
};

// From XOP1WinCustom.rc for Windows.
1100 XOPC                   // Describes operations the XOP adds to Igor.
BEGIN
    "XOP1\0",                               // Name of operation.
    XOPOp | utilOp | compilableOp,          // Operation's category.
    "\0"                                    // NULL required to terminate resource.
END

// From XOP1.c for Macintosh and Windows.

static int
ExecuteXOP1(XOP1RuntimeParamsPtr  p)
{
    int result;
    result = <XOP-specific code to implement command line operation>;
    return result;
}

static int
RegisterXOP1(void)
{
    char* cmdTemplate;
    cmdTemplate = "XOP1 wave";
    return RegisterOperation(cmdTemplate,  . . ., ExecuteXOP1, . . .)
}

static void
XOPEntry(void)
{
    long result = 0;
    switch (GetXOPMessage()) {
        // Handle message
    }
    SetXOPResult(result);
}

void
main(IORecHandle ioRecHandle)
{
    XOPInit(ioRecHandle);
    SetXOPEntry(XOPEntry);
    RegisterXOP1();
    SetXOPResult(0L);
}
```

When the user invokes an XOP's external operation or external function, in most cases Igor directly calls a function which was registered by the XOP for that operation or function. Chapters 5 and 6 discuss the details of creating external operations and functions.

Except for calling an external operation or an external function, all communications from Igor to the XOP go through the XOPEntry routine. For example, if the XOP adds a menu item to an Igor menu, when the user chooses the menu item Igor calls XOPEntry, passing the MENUITEM message and parameters in fields of the IORecHandle. The XOPEntry routine then calls GetXOPMessage to determine what message Igor is sending and GetXOPItem to obtain parameters associated with the message.

The IORecHandle is managed completely by the XOPSupport library routines. Except for passing the IORecHandle to the XOPInit function during initialization, you can completely ignore it.

# XOP Resources

There are three types of resources that you can use in your XOP:

| Type of Resource | Defined By | Examples |
|---|---|---|
| XOP-specific | WaveMetrics | XOPI, XOPC, XOPF |
| Platform-specific | Apple, Microsoft | Menus, dialogs |
| Your own resources | You | Whatever you define |

The XOP-specific resources describe your XOP to Igor. This section discusses the form and meaning of these resources. This table lists all of the XOP-specific resources:

| Resource | What It Does | Explained In |
|---|---|---|
| XOPI 1100 | Describes XOP's general properties to Igor. | This chapter |
| XOPC 1100 | Defines operations added by the XOP. | Chapter 5 |
| XOPF 1100 | Defines functions added by the XOP. | Chapter 6 |
| STR# 1100 | Defines error messages added by the XOP. | Chapter 10 |
| STR# 1101 | Miscellaneous XOP strings that Igor needs to know about. | Chapter 8 |
| STR# 1160 | Used by XOPs that add target windows. | Chapter 9 |
| XMN1 1100 | Defines main menu bar menu added by XOP. | Chapter 8 |
| XMI1 1100 | Defines menu items added to Igor menus. | Chapter 8 |
| XSM1 1100 | Defines submenus used by XOP. | Chapter 8 |

The STR# resource is a standard Apple resource format. The other listed resource types are custom XOP resource formats.

All XOPs must have an XOPI resource. Igor will refuse to run your XOP if it lacks this. All of the other resources are required only if the XOP uses the associated features.

## Creating Resources on Macintosh

You can create Macintosh resources by editing a .r file as text and then letting your development system compile the .r file or by using a resource editor such as ResEdit or Resorcerer. For the XOP-specific resources, you must edit the .r file. You should start with a .r file from a WaveMetrics sample XOP.

With Mac OS X, Apple has moved away from using resources to describe user interface items like menus, controls, dialogs and windows. Consequently it has abandoned ResEdit. In its place, Apple has created “nibs”, which are files containing descriptions of user interface items, and the Interface Builder application, which you can use to create nibs.

XOP Toolkit 5 and Igor Pro 5 do not use or support nibs. You can use nibs and Interface Builder for user interface items that Igor does not need to know about, such as controls in windows you create or dialogs that you run with your own dialog code.

For those items that Igor does need to know about, such as the XOP-specific resources and menus, you must edit a .r file. Macintosh development systems include resource compilers that they use to compile any .r files in the project. The resulting resources are stored in the resource fork of CFM XOPs and in the .rsrc file in the package folder of Mach-O XOPs.

The XOPTypes.r XOPSupport file defines format of the custom WaveMetrics resource types such as XOPI, XOPC and XOPF. XOPTypes.r is #included in your .r file.

## Creating Resources on Windows

On Windows, Visual C++ creates resources by compiling .rc files. You can create the .rc file either by entering text directly into the file or by using the development system's resource editor.

On Windows, the XOP-specific resources are stored in a separate .rc file which is edited as a text file. For example, the XOP1 sample XOP includes an XOP1.rc file and an XOP1WinCustom.rc file. XOP1WinCustom.rc contains the XOP-specific resources. XOP1.rc contains any other resources (e.g., version and menu resources) and also contains an include statement that includes the XOP1WinCustom.rc.

To include the XOP1WinCustom.rc file in the XOP1.rc file, use the Resource Includes dialog. Visual C++ will not let you invoke the Resource Includes dialog until you have inserted at least one resource into the project. If you do not already have a standard resource file, like XOP1.rc, you can create one by creating a version resource for your project. Once you have added a standard resource file to the project, you must display the resource view of the project and select the folder icon that represents the project resources. Now the Resource Includes item in the View menu (Visual C++ 6) or Edit menu (Visual C++ 7) will be enabled, and you can invoke the Resource Includes dialog.

In Visual C++ 6, choose View->Resource Includes. In Visual C++ 7, choose Edit->Resources View. In both systems, enter the include statement in the Compile-time Directives list of the Resource Includes dialog:

```
#include "XOP1WinCustom.rc"
```

When you close the Resource Includes dialog, Visual C++ may display a warning saying "Directive text will be written verbatim into your resource script and may render it uncompilable". Despite this dire warning, you should click OK and then click OK in the Resource Includes dialog to include the custom file.

When you change a resource using the built-in resource editor or change the compile-time directives, Visual C++ writes out a new version of the main .rc file (e.g., XOP1.rc), which contains the include statement referencing the XOP-specific .rc file (e.g., XOP1WinCustom.rc). Visual C++ also generates a file named resource.h, which is also included in the main .rc file.

## The XOPI 1100 Resource

There is one resource that must be present in every XOP. That is the XOPI 1100 resource. This resource tells Igor a few things about your XOP. The form of the resource is as follows.

```
// Macintosh
#include "XOPStandardHeaders.r"  // Defines XOP-specific types and symbols.
type 'XOPI' {            // XOPI - Describes general XOP properties to Igor.
   XOP_VERSION,          // XOP protocol version.
   DEV_SYS_CODE,         // Code for development system used to make XOP.
   0,                    // Obsolete - set to zero.
   0,                    // True if XOP requires math coprocessor
   XOP_TOOLKIT_VERSION, // Version of XOP Toolkit used to create XOP.
};

// Windows
#include "XOPResources.h"  // Defines symbols used below.
1100 XOPI                // XOPI - Describes general XOP properties to Igor.
BEGIN
   XOP_VERSION,          // XOP protocol version.
   DEV_SYS_CODE,         // Code for development system used to make XOP.
   0,                    // Obsolete - set to zero.
   0,                    // Obsolete - set to zero.
   XOP_TOOLKIT_VERSION, // Version of XOP Toolkit used to create XOP.
END
```

If the XOPI 1100 resource is missing then Igor will not run the XOP.

The symbols XOP_VERSION, DEV_SYS_CODE and XOP_TOOLKIT_VERSION are defined in XOPResources.h which is included by XOPStandardHeaders.r. The values of these macros are automatically set; you don't need to change them.

The first field is used to make sure that the XOP is compatible with the XOP interface in the version of Igor that is running. This is the version number of Igor's XOP protocol at the time the XOP was compiled, defined by the XOP_VERSION constant in the XOPResources.h file. See **Igor/XOP Compatibility Issues** on page 140 for details.

The next field tells Igor what development system was used to compile the XOP, which in some cases determines aspects of how Igor calls the XOP.

The next two fields are obsolete and must be set to zero.

The final field tells Igor what version of the XOP Toolkit was used to compile the XOP, which in some cases determines aspects of how Igor calls the XOP. Prior to XOP Toolkit 5, this field had a different meaning but always had the value 0 or 1. With XOP Toolkit 5/Igor Pro 5s, the field was pressed into different service in a backward-compatible way

# Basic XOP Messages

Here are the basic messages that Igor can send to your XOP's XOPEntry routine. The constants (INIT, IDLE, CMD, etc.) are defined in XOP.h. When Igor calls your XOPEntry routine, it calls the GetXOPMessage XOPSupport routine to determine what message is being set. The arguments listed are arguments Igor is passing to you. You access them using the GetXOPItem XOPSupport routine.

**NOTE**: You must get the message from Igor and get all of the arguments for the message before you do any callbacks to Igor. For details see **Avoiding Common Pitfalls** on page 320.

A typical XOP will ignore most of these messages and respond to just a few of them. An XOP that just adds external operations or functions may ignore all of them.

As part of the response to a message from Igor, your XOP can pass a result code back to Igor using the SetXOPResult XOPSupport routine. The "Result" item in the following descriptions refers to this result. See **XOP Errors** on page 127 for details on the error codes that you can return to Igor.

Some messages have arguments of type Rect* (pointer to Rect). This is a pointer to a Macintosh rectangle structure, even when running on Windows.

**INIT**

Tells your XOP that it needs to initialize itself.

Arguments: None.
Result: Error code from your initialization.

This is the first message received by every XOP when it is loaded. This message will be received by your main routine. Subsequent messages will be received by your XOPEntry routine.

If your XOP returns a non-zero result via the SetXOPResult XOPSupport routine, Igor will dispose of it immediately. Therefore you should do any necessary cleanup before returning a non-zero result.

**IDLE**

Tells your XOP to do its idle processing.

Arguments: None.
Result: None.

Your XOP gets this message if it has set the IDLES bit using the SetXOPType XOPSupport routine. You should do this if your XOP has tasks that it needs to do periodically so that you'll get the periodic IDLE message.

**MENUITEM**

Tells your XOP that its menu item has been selected.

Argument 0: Menu ID of menu user selected.
Argument 1: Item number of item user selected.
Result: Your result code from operation.

Your XOP must determine which of its menu items has been invoked and respond accordingly. It should call SetXOPResult to pass zero or an error code back to Igor.

See Chapter 8, **Adding Menus and Menu Items**, for details on how to respond to this message.

**MENUENABLE**

Tells your XOP to enable or disable its menu items.

Arguments: None.
Result: None.

If your XOP has one or more menu items, it receives this message when the user clicks in the menu bar or presses a command-key equivalent (*Macintosh*) or accelerator (*Windows*). This message gives your XOP a chance to enable, disable or change its menu items as appropriate.

As of Igor Pro 5, an XOP receives the MENUENABLE message if its window is the front window, even if the XOP adds no menu items to Igor. Previously Igor sent the MENUENABLE message only to XOPs that added a menu item or a menu.

See Chapter 8, **Adding Menus and Menu Items**, for details on how to respond to this message.

**CLEANUP**

Tells your XOP that it is about to be closed and discarded from memory.

Arguments: None.
Result: None.

Your XOP should dispose any memory blocks that it has allocated, close any windows that it has opened and do any other necessary cleanup.

**OBJINUSE**

Allows your XOP to tell Igor that it is using a particular object.

| | |
|---|---|
| Argument 0: | Object identifier. |
| Argument 1: | Object type. |
| Result: | Zero if your XOP is not using the object, one if it is using the object. |

Igor passes this message to your XOP when it is about to dispose of an object (e.g., kill a wave) that your XOP may depend upon.

At present, the only use for this is to tell Igor that your XOP is using a particular wave so that Igor will not let the user kill the wave. The object identifier is the wave handle. The object type is WAVE_OBJECT. Object types are defined in IgorXOP.h.

You need to respond to this message if your XOP depends on a particular wave's existence. Otherwise, ignore it.

When a new experiment is loaded or Igor quits, all objects are killed, whether they are in use or not. Therefore, if your XOP expects one or more Igor objects to exist, the XOP must be prepared to stop using the objects when either of these events occurs. Igor calls your XOP with the NEW message when a new experiment is about to be opened and with the CLEANUP message when Igor is about to quit.

**FUNCADDRS**

Asks your XOP for the address of one of its functions.

| | |
|---|---|
| Argument 0: | Function index number starting from 0. |
| Result: | Address of function or NULL. |

See Chapter 6, **Adding Functions**, for details.

# Messages for Operations and Functions

Most XOPs that add operations or functions will be set up such that Igor will call a specific C function in the XOP when the user invokes the XOP's external operation or function. This is called the "direct" method. Such XOPs will not receive the messages discussed in this section. Direct external operations were added in Igor Pro 5. Both Igor Pro 4 and Igor Pro 5 support direct external functions.

These messages are sent only to XOPs that use the "message" method to implement external operations and functions.

**CMD**

Tells your XOP to parse and execute one of the operations that it added to Igor. If your XOP uses Operation Handler (described on page 151), it will not receive this message.

Argument 0: Pointer to C string containing name of operation.
Argument 1: Index identifying the operation being invoked.
Result: Your result code from the operation.

Your XOP must determine which of its operations has been invoked, parse the operation's parameters and execute the operation. It must call SetXOPResult to pass zero or an error code back to Igor.

The operation index is the number, starting from zero, of the operation in the XOP's XOPC 1100 resource. This resource, described in Chapter 5, lists each of the operations added by the XOP.

See Chapter 5, **Adding Operations**, for details.

**FUNCTION**

Tells your XOP to execute one of its functions.

Argument 0: Function index number starting from 0.
Result: Function error code or 0.

Your XOP must determine which of its functions has been invoked and execute the function. It should call SetXOPResult to pass zero or an error code back to Igor.

The function index is the number, starting from zero, of the function in the XOP's XOPF 1100 resource. This resource, described in Chapter 6, lists each of the functions added by the XOP.

See Chapter 6, **Adding Functions**, for details.

# Messages for XOPs with Windows

If your XOP has one or more windows of its own you will need to respond to some of the following messages from Igor. Many of these messages work hand-in-hand with callbacks for XOPs with text windows.

For a simple example of adding a window to Igor, see the WindowXOP1 sample XOP. For a more complex example, see the TUDemo sample.

Some of the window-related messages are sent to Macintosh XOPs only. These are noted below. Windows XOPs receive analogous messages directly from the Windows OS. See Chapter 9 for details.

Many of the window-related messages include an argument that refers to a window. The type of this argument is XOP_WINDOW_REF. XOP_WINDOW_REF is defined in XOP.h. On Macintosh, it is a WindowPtr. On Windows, it is an HWND.

**ACTIVATE**

Tells your XOP that it needs to activate or deactivate a window.

Argument 0: XOP_WINDOW_REF for window to activate.
Argument 1: Modifiers field from activate EventRecord.
Result: none.

Windows Note: Igor does not send the ACTIVATE message to Windows XOPs. Instead, the XOP's window procedure receives the WM_MDIACTIVATE message directly from the Windows OS.

**UPDATE**

Tells your XOP that it needs to update a window.

Argument 0: XOP_WINDOW_REF for window to update.
Result: None.

Windows Note: Igor does not send the UPDATE message to Windows XOPs. Instead, the XOP's window procedure receives the WM_PAINT message directly from the Windows OS.

**GROW**

Tells your XOP that it needs to change the size of its window.

Argument 0: XOP_WINDOW_REF for the window.
Argument 1: Size to set the window to.
Result: None.

The size argument has the vertical size for the window in its high word and the horizontal size for the window in its low word. However, if size is zero it means you need to zoom your window in or out.

Windows Note: Igor does not send the GROW message to Windows XOPs. Instead, the XOP's window procedure receives the WM_SIZE message directly from the Windows OS.

**SETGROW**

Asks your XOP for the minimum allowable size for window.

| | |
|---|---|
| Argument 0: | XOP_WINDOW_REF. |
| Argument 1: | Pointer to a Rect to receive grow limits. |
| Result: | None. |

Set the top coordinate of the Rect to the minimum vertical size in pixels for your window. Set the left coordinate to the minimum horizontal size in pixels for your window. Leave the right and bottom coordinates alone. They are set by Igor based on the current screen setup.

Igor uses the values returned via the Rect pointer to set a lower limit on the size of the window when the user resizes the window on Macintosh. On Windows, the values returned have no effect.

**CLOSE**

Tells your XOP that the user wants to close its window.

| | |
|---|---|
| Argument 0: | XOP_WINDOW_REF. |
| Argument 1: | Close type (Igor Pro 2.0 or later). |
| Result: | None. |

Igor sends the CLOSE message when the user chooses Close from the Windows menu. On Macintosh, Igor also sends the close message when the user clicks the close box. On Windows, when the user clicks the close button, the Windows OS sends a WM_CLOSE message directly to the XOP window procedure and Igor does not send the CLOSE message.

Depending on your XOP, closing the window may also close the XOP or may just hide the window. It’s up to you.

The close type argument has the following meaning:

| | |
|---|---|
| 0 | Normal close (ask about saving and then close). |
| 1 | Close without saving. |
| 2 | Save without asking and close. |

The argument will have the value 1 if the user presses option while clicking your window’s close box on Macintosh.

**CLICK**

Tells your XOP a click occurred in its window.

Argument 0: XOP_WINDOW_REF.
Argument 1: Pointer to EventRecord describing click.
Result: None.

Service or ignore the click event as appropriate.

Windows Note: Igor does not send the CLICK message to Windows XOPs. Instead, the XOP's window procedure receives the WM_LBUTTONDOWN and WM_RBUTTONDOWN messages directly from the Windows OS.

**KEY**

Tells your XOP a keydown event occurred in its window.

Argument 0: XOP_WINDOW_REF.
Argument 1: Pointer to EventRecord describing key.
Result: None.

Service or ignore the keydown as appropriate.

Windows Note: Igor does not send the KEY message to Windows XOPs. Instead, the XOP's window procedure receives the WM_KEY and WM_CHAR messages directly from the Windows OS.

**NULLEVENT**

Tells your XOP a null event occurred while its window was active.

Argument 0: XOP_WINDOW_REF.
Argument 1: Pointer to EventRecord describing null event.
Result: None.

Set the cursor based on the where field in the EventRecord.

Windows Note: Igor does not send the NULLEVENT message to Windows XOPs. Instead, the XOP's window procedure receives the WM_MOUSEMOVE message directly from the Windows OS.

**WINDOW_MOVED**

Tells you that the user moved your window by dragging or by using the MoveWindow command line operation.

Argument 0: XOP_WINDOW_REF identifying your window.
Result: None

Windows Note: Igor does not send the WINDOW_MOVED message to Windows XOPs. Instead, the XOP's window procedure receives the WM_MOVE message directly from the Windows OS.

**MOVE_TO_PREFERRED_POSITION**

Tells your XOP that the user wants to move it to its preferred position. Igor sends this message when the user chooses Move To Preferred Position from Igor's Window Control submenu.

| | |
|---|---|
| Argument 0: | XOP_WINDOW_REF identifying your window. |
| Result: | None. |

If your window has a preferred or default size and position, you can respond to this message by moving it to that size and position.

If you want to support this message, you must enable the Move to Preferred Position menu item by calling SetIgorMenuItem, as described in Chapter 8.

Windows Note: If the window is maximized or minimized when the user chooses this menu item, Igor will restore the window to its normal state before sending this message to your XOP.

**MOVE_TO_FULL_POSITION**

Tells your XOP that the user wants to move it to its "full size position". The meaning of "full size position" depends on the window. It may be the size and position that displays all of the window's content or that fills the Macintosh screen or Windows MDI frame. Igor sends this message when the user chooses Move To Full Size Position from Igor's Window Control submenu.

| | |
|---|---|
| Argument 0: | XOP_WINDOW_REF identifying your window. |
| Argument 1: | Pointer to a suggested rectangle (Rect*). |
| Result: | None. |

If you just want to fill the Macintosh screen or the MDI frame, you can use the suggested rectangle passed by Igor as argument 0. It is in units of pixels and defines the content region of the window - that is, the region of the window excluding the window title bar or caption and the window border, if any. You can move the window to the suggested position by calling TransformWindowCoordinates and then SetXOPWindowIgor-PositionAndState.

If you want to be fancy, you can calculate your own rectangle that is customized for your window's content.

If you want to support this message, you must enable the Move to Full Size Position menu item by calling SetIgorMenuItem, as described in Chapter 8.

Windows Note: If the window is maximized or minimized when the user chooses this menu item, Igor will restore the window to its normal state before sending this message to your XOP.

**RETRIEVE**

Tells your XOP that the user wants to move your window entirely on screen (Macintosh) or entirely within the MDI frame (Windows). Igor sends this message when the user chooses Retrieve Window or Retrieve All Windows from Igor's Window Control submenu.

Argument 0: XOP_WINDOW_REF identifying your window.
Argument 1: Pointer to a suggested rectangle (Rect*).
Result: None.

The suggested rectangle should be appropriate for most if not all cases. It is in units of pixels and defines the content region of the window - that is, the region of the window excluding the window title bar or caption and the window border, if any. You can move the window to the suggested position by calling TransformWindowCoordinates and then SetXOP-WindowIgorPositionAndState.

If you want to support this message, you must enable the Retrieve Window menu item by calling SetIgorMenuItem, as described in Chapter 8.

You will also receive this message if the user chooses the Retrieve All Windows menu item. This is true whether or not you have enabled the Retrieve Windows menu item. If you don't care about this message, you can just ignore it.

Windows Note: Igor will not send this message to your XOP if its window is maximized.

**CUT**

Tells your XOP that the user wants to do a cut in its window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**COPY**

Tells your XOP that the user wants to do a copy in its window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**PASTE**

Tells your XOP that the user wants to do a paste in its window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**CLEAR**

Tells your XOP that the user wants to do a clear in its window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**UNDO**

Tells your XOP that the user wants to do an undo in its window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**FIND**

Tells your XOP that the user wants to do a find in its window.

Argument 0: XOP_WINDOW_REF.
Argument 1: Code for type of find.
(1 for normal, 2 for find same, 3 for find selection)
Result: None.

**REPLACE**

Tells your XOP that the user wants to do a replace in its window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**INDENTLEFT**

Tells your XOP that the user wants to do an indent-left in its window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**INDENTRIGHT**

Tells your XOP that the user wants to do an indent-right in its window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**DISPLAYSELECTION** Tells your XOP that the user wants to see the selected part of its window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**PAGESETUP** Tells your XOP that the user wants to do a page setup for its window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**PRINT** Tells your XOP that the user wants to print all or part of its window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**INSERTFILE** Tells your XOP that the user wants to insert the contents of a file in its window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**SAVEFILE** Tells your XOP that the user wants to save the contents of its window in a file.

Argument 0: XOP_WINDOW_REF.
Result: None.

Igor 1.2 sent this message if the user chose Edit->Save Text while your XOP's window is active. Igor Pro has no Edit->Save Text item and therefore will not send this message.

**DUPLICATE** Tells your XOP that the user wants to do a duplicate in your window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**EXPORT_GRAPHICS** Tells your XOP that the user wants to do an export graphics operation.

Argument 0: XOP_WINDOW_REF.
Result: None.

**SELECT_ALL** Tells your XOP that the user wants to do a select-all in your window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**SAVE_WINDOW** Tells your XOP that the user wants to do a save of your window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**SAVE_WINDOW_AS** Tells your XOP that the user wants to do a save-as of your window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**SAVE_WINDOW_COPY** Tells your XOP that the user wants to do a save-a-copy of your window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**REVERT_WINDOW** Tells your XOP that the user wants to do a revert of your window.

Argument 0: XOP_WINDOW_REF.
Result: None.

**GET_TARGET_WINDOW_NAME**
This message is used by XOPs that add target window types to Igor. See **Adding XOP Target Windows** on page 257 for further information.

**SET_TARGET_WINDOW_NAME**
This message is used by XOPs that add target window types to Igor. See **Adding XOP Target Windows** on page 257 for further information.

**GET_TARGET_WINDOW_REF**
This message is used by XOPs that add target window types to Igor. See **Adding XOP Target Windows** on page 257 for further information.

**SET_TARGET_WINDOW_TITLE**
This message is used by XOPs that add target window types to Igor. See **Adding XOP Target Windows** on page 257 for further information.

**SAVE_WINDOW_MACRO**
This message is used by XOPs that add target window types to Igor. See **Adding XOP Target Windows** on page 257 for further information.

# Messages for XOPs that Save and Load Settings

An elaborate XOP can save settings and/or documents as part of an Igor experiment and load those settings and documents when the user reopens the experiment. To do this, your XOP needs to respond to the messages in this section.

**NEW**

Tells your XOP that the user selected New Experiment from the File menu.

Arguments: None.
Result: None.

When the user creates a new experiment your XOP should set its settings to their default state.

**MODIFIED**

Asks your XOP if it has been modified since the last NEW, LOAD or SAVE message and needs to save settings or documents.

Arguments: None.
Result: 0 if your XOP is not modified, 1 if modified.

You need to respond to this message if your XOP saves settings or documents as part of an experiment. Otherwise, ignore it.

**CLEAR_MODIFIED**

Igor sends this message during its initial startup, and after Igor generates a new experiment (in response to the user choosing New Experiment), and after Igor finishes loading an experiment (in response to the user choosing Open Experiment or Revert Experiment), and after Igor finishes saving an experiment (in response to the user choosing Save Experiment or Save Experiment As). An XOP that saves data in experiment files can use this message to clear its modified flag.

Arguments: None.
Result: None.

**SAVE**

This message is obsolete. Igor Pro will not send it.

**LOAD**

This message is obsolete. Igor Pro will not send it.

**SAVESETTINGS**

Allows your XOP to save settings as part of an experiment.

Argument 0: Save type.
Argument 1: Experiment type.
Result: Handle to your XOP's settings or NULL.

Your XOP can save its settings in the experiment file. If you want to do this, return a handle to the settings data to be saved. Otherwise, ignore the message. See **Saving and Loading XOP Settings** on page 133 for details.

If you return a handle, Igor stores the data in the experiment file and disposes the handle. Don't access the handle once you pass it to Igor.

The save type will be one of the following which are defined in IgorXOP.h:

SAVE_TYPE_SAVE
SAVE_TYPE_SAVEAS
SAVE_TYPE_SAVEACOPY
SAVE_TYPE_STATIONERY

The experiment type will one of the following which are defined in IgorXOP.h:

EXP_PACKED
EXP_UNPACKED

**LOADSETTINGS**

Allows your XOP to load settings it saved in an experiment.

| | |
|---|---|
| Argument 0: | Handle to data saved in experiment file or NULL. |
| Argument 1: | Load type. |
| Argument 2: | Experiment type. |
| Result: | None. |

If your XOP saved its settings in the experiment being opened you should reload those settings from the handle. If your experiment did not save its settings then the handle will be NULL and you should do nothing. See **Saving and Loading XOP Settings** on page 133 for details.

The handle that Igor passes to you comes from the experiment file and will be automatically disposed when Igor closes the experiment file so don't access it once you finish servicing the LOADSETTINGS message.

The load type will be one of the following which are defined in IgorXOP.h:

LOAD_TYPE_OPEN

LOAD_TYPE_REVERT

LOAD_TYPE_STATIONERY

LOAD_TYPE_MERGE (Igor Pro 5 or later)

The experiment type will one of the following which are defined in IgorXOP.h:

EXP_PACKED

EXP_UNPACKED

# XOP Errors

The XOP protocol facilitates providing helpful error messages when something goes wrong in your XOP. You can return an error code to Igor in response to the INIT, CMD, FUNCTION and MENUITEM messages. You pass the error code to Igor by calling the SetXOPResult XOPSupport routine. When you do this, Igor displays an error dialog. Igor displays the dialog when your XOP returns, not immediately.

Igor will also display an error message if you return a non-zero result from a direct external operation or direct external function.

In some cases, you may need to display an error dialog other than in response to these messages. In this case you need create your own error dialog routine or use the IgorError XOPSupport routine. See **Displaying Your Own Error Alert** on page 131 for details on how to do this.

The GetIgorErrorMessage XOPSupport routine returns an error message for a specified error code. This is useful if you want to display an error message in your own window.

## Error Codes

There are three kinds of errors that you can return to Igor: Igor error codes (in the range -1 to 9999), Mac OS error codes (in the range -32768 to -2) and your own custom error codes (in the range 10000 to 10999). You must not return Windows OS error codes to Igor because these codes conflict with Igor error codes. Instead, you must translate Windows OS Error codes into Igor error codes. This is explained below.

## Igor Error Codes

An Igor error code is a code for an error that Igor might generate itself. The file IgorXOP.h contains #defines for each of the Igor error codes. Your XOP can return an Igor error code and Igor will display an appropriate error dialog. For example, if your XOP returns NOMEM, Igor displays an "out of memory" error dialog and if your XOP returns NOWAV, Igor displays an "expected wave name" error dialog.

In the IgorXOP.h file you will find

```
#define FIRST_IGOR5_ERR 808
```

This marks the start of error codes added in Igor Pro 5. If you want your XOP to run under Igor Pro 4, you can not return error codes greater than or equal to FIRST_IGOR5_ERR. Instead, use custom error codes.

There is one special Igor error code: -1. It signifies an error for which an error dialog is not necessary. Return -1 if the user aborts an operation or if you reported the cause of the error in some other way. This tells Igor that an error occurred but Igor will not display an error dialog.

## XOP Custom Error Codes

Custom XOP error codes are defined by an XOP. #defines for these error codes should be created in the XOP's .h file and there must be a corresponding error message in the XOP's STR# 1100 resource which is described in Chapter 10. Custom error codes are numbered starting from FIRST_XOP_ERR which is defined in XOP.h. The details of adding custom error messages are discussed on page 130 under **Adding Custom Errors**.

## Mac OS Error Codes

A Mac OS error code is a code for an error generated by the Macintosh operating system. A Mac OS error code is always negative. Igor contains tables that allow it to display meaningful error messages for the most common Mac OS error codes. For other Mac OS errors, Igor displays a generic error message.

## Handling Windows OS Error Codes

Windows OS error codes conflict with Igor error codes. For example, the error code 1 (NOMEM) means "out of memory" in Igor. On Windows, it means "The function is invalid" (ERROR_INVALID_FUNCTION). Igor interprets error codes that you return to it as Igor error codes. If you pass a Windows OS error code to Igor, you will get an incorrect error message.

On Windows, Igor provides two functions to deal with this problem. WindowsErrorToIgorError translates a Windows error code that you get from the Windows GetLastError function into an error code that Igor understands. WMGetLastError is a WaveMetrics substitute for the Windows GetLastError function that also returns error codes that Igor understands.

Here is an example illustrates how you might use WindowErrorToIgorError.

```
int
Test(void)        // Returns 0 if OK or a non-zero error code.
{
   int err;

   if (SetCurrentDirectory("C:\\Test") == 0) {
      err = GetLastError();
      // You can add code to examine err here, if necessary.
      err = WindowsErrorToIgorError(err);
      return err; // Return code to Igor.
   }
   return 0;
}
```

This is the recommended technique if your own code needs to examine the specific error code returned by GetLastError. Usually, the precise error code that GetLastError returns does not affect the flow of your code. In such cases, we can rewrite the function using WMGetLastError, which itself calls GetLastError and WindowsErrorToIgorError.

```
int
Test(void)        // Returns 0 if OK or a non-zero error code.
{
   int err;

   if (SetCurrentDirectory("C:\\Test") == 0) {
      err = WMGetLastError();
      return err; // Return code to Igor.
   }
   return 0;
}
```

In addition to translating the error code, WMGetLastError is different from GetLastError in two regards. First, it always returns a non-zero result. Second, it calls SetLastError(0) after it calls GetLastError. The following example illustrates why WMGetLastError does these things.

```
/* GetAResource(hModule, resID, resType)

   Returns 0 if OK or non-zero error code.
*/
int
GetAResource(HMODULE hModule, int resID, const char* resType)
{
   HRSRC hResourceInfo;

   hResourceInfo=FindResource(hModule,MAKEINTRESOURCE(resID),resType);
   if (hResourceInfo == NULL) {  // hResourceInfo is NULL
      err = GetLastError();      // but GetLastError returns 0.
      return err;    // So we return 0 but hResourceInfo is NULL.
   }
   .
   .
   .
}
```

The Windows documentation for FindResource says "If the function fails, the return value is NULL. To get extended error information, call GetLastError.

However, empirical evidence under Windows 95 indicates that, if the resource is not found, FindResource returns NULL, but *does not* set the last error. Therefore, GetLastError returns a stale error code - whatever was set by some previous Windows API call. If this happens to be zero, then the function above will return zero and the calling routine will think that it succeeded, with possibly disastrous results.

If we use WMGetLastError instead of GetLastError, we will always get a non-zero result, avoiding the disaster. WMGetLastError calls GetLastError and, if GetLastError returns 0, WMGetLastError returns WM_UNKNOWN_ERROR. Furthermore, since WMGetLastError

calls SetLastError(0) after calling GetLastError, we can be certain that we will not get a stale error code the next time we call it.

You should call GetLastError or WMGetLastError only when you have received another indication of failure from a Windows API call. For example, FindResource signals a failure by returning NULL. You should not call GetLastError or WMGetLastError if FindResource returns non-NULL. This is because Windows API routines do not set the last error code if they succeed. They set it only if they fail, and sometimes not even then.

## Adding Custom Errors

Each time your XOPEntry routine receives the INIT, CMD, FUNCTION or MENUITEM message, you must return an error code to Igor using the SetXOPResult XOPSupport routine. Also, your direct external operations and functions must return an error code as its function result. If the error code is non-zero, Igor will display an error dialog when your XOP returns.

Custom XOP error codes are codes that you define for your XOP, typically in a .h file of your own. For each code, you define a corresponding error message. You store the error messages in your STR# 1100 resource in the resource fork of your XOP.

The custom error codes for your XOP must start from FIRST_XOP_ERR+1. FIRST_XOP_ERR is defined in the XOP.h file. As an illustration, here are the custom error codes for the XFUNC3 sample XOP as defined in XFUNC3.h.

```
#define REQUIRES_IGOR_200 1 + FIRST_XOP_ERR
#define UNKNOWN_XFUNC 2 + FIRST_XOP_ERR
#define NO_INPUT_STRING 3 + FIRST_XOP_ERR
```

Here is XFUNC3’s STR# 1100 resource, which defines the error messages corresponding to these error codes.

```
// Macintosh, in XFUNC3.r.
resource 'STR#' (1100) {   // Custom error messages
   {
      "XFUNC3 requires Igor Pro 2.0 or later.",
      "XFUNC3 XOP was called to execute an unknown function.",
      "Input string is non-existent.",
   }
};

// Windows, in XFUNC3WinCustom.rc.
1100 STR#
BEGIN
   "XFUNC3 requires Igor Pro 2.0 or later.\0",
   "XFUNC3 XOP was called to execute an unknown function.\0",
   "Input string is non-existent.\0",
   "\0"        // NULL required to terminate the resource.
END
```

## Displaying Your Own Error Alert

Igor displays an error alert if you report an error in response to the INIT, CMD, FUNCTION and MENUITEM messages or as the function result of a direct operation or function. You can call the IgorError XOPSupport routine to display an alert at other times. IgorError accepts an Igor, Mac OS, or XOP-defined error code and displays a dialog with the corresponding message.

You may want to display an error message in your own dialog or window. In this case, call GetIgorErrorMessage. GetIgorErrorMessage accepts an Igor, Mac OS, or XOP-defined error code and returns a string containing the corresponding message.

To display your own alert using a string that is not associated with an error code, you can use the XOPOKAlert, XOPOKCancelAlert, XOPYesNoAlert, and XOPYesNoCancelAlert XOPSupport routines.

# XOPs and Preferences

If you are writing an XOP that has a user interface, you may want to save certain bits of information so that the user does not need to re-enter them each time he or she uses the XOP. For example, the GBLoadWave XOP saves the state of many of its dialog items and restores them the next time the user invokes the dialog.

The SaveXOPPrefsHandle XOPSupport routine saves your XOP's data in Igor preferences. You can later retrieve this data by calling the GetXOPPrefsHandle XOPSupport routine. As of this writing the data is stored in the Igor preferences file but you can not count on this. A future version of Igor may store your data in a separate file.

Each time you call either of these routines, the Igor preferences file is opened and closed. Therefore it is best to call each of them only once. One way to do this is to call GetXOPPrefsHandle when your XOPs starts and SaveXOPPrefsHandle when you receive the CLEANUP message.

The GetPrefsState callback provides a way for you to know if preferences are on or off. This allows an XOP to behave like Igor in regard to when user preferences are used versus when factory default preferences are used, as described in the Igor Pro manual.

The data that you store will usually be in the form of a structure. As time goes by, you may want to add fields to the structure. In that case, you have the problem of what happens if an old version of your XOP receives the new structure. The easiest way to deal with this is to define an ample amount of reserved space in very first incarnation of your structure. Set all of the reserved space to zero. In future versions you can use some of the reserved space for new fields with the value zero representing the default value. Old versions of your XOP will ignore the new fields.

# XOPs and Experiments

If you develop an elaborate XOP, you might want it to store settings or documents as part of an Igor experiment. Igor provides help in doing this.

When the user creates a new experiment or opens an existing experiment, Igor sends you a NEW message. You should reset your XOP settings to their default states. When the user opens an experiment, Igor sends the LOADSETTINGS message. When the user saves an experiment, Igor sends the SAVESETTINGS message.

Prior to Igor Pro 4, Igor also sent LOAD and SAVE messages when the user opened or saved an unpacked experiment. These messages were not suitable for Windows or OS X and are no longer sent.

## Saving and Loading XOP Settings

If your XOP has settings that it wants to save in each experiment then you need to respond to the SAVESETTINGS and LOADSETTINGS messages. See the XOPSaveSettings and XOPLoadSettings routines in VDT2:VDT.c for an example of responding to these messages.

When the user saves an experiment, Igor sends your XOP the SAVESETTINGS message. If you want to save settings in the experiment, you need to make a handle containing the settings. Return this handle to Igor using the SetXOPResult routine. Once you've passed this handle back to Igor, it belongs to Igor, so don't access, modify, or delete it.

Your data is stored in a record of a packed experiment or in the miscellaneous file inside the home folder of an unpacked experiment. You should not attempt to access the data directly since the method by which it is stored could be changed in future versions of Igor.

When the user opens an experiment, after sending the NEW message, Igor sends the LOADSETTINGS message. The primary argument to this message, which you access using the GetXOPItem call, is the handle to the settings that you saved in response to the SAVESETTINGS message or NULL if you saved no settings in the experiment being opened. If it is not NULL, use this handle to restore your XOP to its previous state. The handle belongs to Igor and Igor will dispose it when you return. If you need to, make a copy of the handle.

Igor sends the LOADSETTINGS message during the experiment recreation process before any of the experiment's objects, such as waves, variables and data folders, have been created. If your settings contain a reference to an Igor object, you will not be able to access the object until the experiment recreation process is complete. For example, if your XOP receives IDLE messages, you can access the object on the first IDLE message after the LOADSETTINGS message. You can also use the CLEAR_MODIFIED message for this purpose.

If your XOP is to run cross-platform, you must take special measures so that you can use settings stored on Macintosh when running on Windows and vice-versa. This is necessary for two reasons. First, unless you take precautions, structures that you declare on one platform will not have the same layout as structures that you declare on the other platform, because of different structure alignment methods. Second, Macintosh and Windows store multi-byte data in the reverse order.

To make sure that your structures have the same layout on both platforms, you need to tell the compiler how you want structures laid out. The sample XOPs on both Macintosh and Windows, are set up to use two-byte alignment for all structures, thus satisfying this condition. For details on this, see **Structure Alignment** on page 279.

If you are loading settings stored on one platform while running on the other, you need to do byte reordering to fix the byte order. Before you can do byte reordering, you need to know what platform the settings come from. You can achieve this by putting a short (two byte) version code into which you store a value from 1 to 255 at the start of your settings structure. This ensures that the high order byte will always be zero and the low order byte will always be non-zero. When you receive the settings handle from Igor, you can examine the version code. If the high order byte is zero, then you do not need to do byte reordering. If it is non-zero, then the settings come from the other platform and you do need to do byte reordering.

The actual byte-reordering of a settings structure is a laborious process of reordering each field in the structure. You can call the XOPSupport FixByteOrder routine once for each field in your structure to achieve this. If this is too much trouble, you can use default settings if you receive settings from the other platform.

The data that you store will usually be in the form of a structure. As time goes by, you may want to add fields to the structure. In that case, you have the problem of what happens if an old version of your XOP receives the new structure. The easiest way to deal with this is to define an ample amount of reserved space in very first incarnation of your structure. Set all of the reserved space to zero. In future versions you can use some of the reserved space for new fields with the value zero representing the default value. Old versions of your XOP will ignore the new fields.

# The IORecHandle

The IORecHandle is a handle to an IORec data structure which contains all of the information that Igor and the XOP need to share. Each XOP has a global variable of type IORecHandle named XOPRecHandle which is defined in XOPSupport.c. The value of XOPRecHandle is set when your XOP's main function calls XOPInit.

You should never deal with the IORecHandle directly since the routines in XOPSupport.c do this for you. However, examining some of the uses of the IORecHandle will help you get a feel for the XOP protocol.

## Resident and Transient XOPs

The **XOPType** field indicates to Igor the capabilities or mode of the XOP. You set this field using the SetXOPType XOPSupport routine. The value of XOPType will be some combination of the following bit-mask constants which are defined in XOP.h.

| Constant | What It Means |
|---|---|
| RESIDENT | XOP wants to remain in memory indefinitely. |
| TRANSIENT | XOP wants to be closed and discarded. |
| IDLES | XOP wants to receive periodic IDLE messages. |
| ADDS_TARGET_WINDOWS | XOP adds a target window type to Igor. |

The XOPType field is inititialized by Igor to RESIDENT. This means that the XOP will stay in memory indefinitely. All XOPs that add direct operations or direct or indirect functions to Igor must be resident. XOPs that perform ongoing services in the background, such as data acquisition XOPs, must also be resident. XOPs that are called very frequently should be resident to avoid the need to repeatedly load and unload them.

In the distant past, when memory was a very scarce commodity, it was advantageous to make an XOP transient if possible. Now, because of the greater availability of memory and because XOPs that add direct operations or functions are not allowed to be transient, it is no longer recommended.

For the rare case where it is wise to make an XOP transient, here is how you do it. Once your operation has done its work, call SetXOPType(TRANSIENT). Shortly thereafter Igor will call your XOP with the CLEANUP message and will then purge your XOP from memory.

### Receiving IDLE Messages For Background Processing

If you want your XOP to receive periodic IDLE messages you need to call

```
SetXOPType(RESIDENT | IDLES)
```

Data acquisition XOPs and other XOPs that provide ongoing background services should make this call.

## Messages, Arguments and Results

The **XOPEntry** field in the IORec structure contains the address of the routine that Igor calls to pass a message to your XOP. The main function in your XOP must set this field so that it points to your XOPEntry routine by calling the SetXOPEntry XOPSupport routine. All subsequent calls from Igor, except for direct operation and direct function calls, go through your XOPEntry routine.

Igor calls your XOPEntry routine, passing no direct parameters. Instead, Igor stores a message code and any arguments that the message requires in the your XOP's IORecHandle. You call the GetXOPMessage XOPSupport routine to determine the message that Igor is sending you. Then, if the message has arguments, you call the GetXOPItem XOPSupport routine to get them. GetXOPMessage and GetXOPItem use the global XOPRecHandle that was set when you called XOPInit from your main function.

Sometimes your response to a message from Igor involves making a call back to Igor to ask it for data or a service. For example, the FetchWave callback asks Igor for a handle to a wave with a particular name and the XOPNotice callback asks Igor to display some text in the history area. When you make a callback, the XOPSupport routines that implement the callback use the same fields in the IORecHandle that Igor used when it passed the original message to you. For this reason, it is imperative that you get the message and any arguments that Igor is sending to you before doing any callbacks.

**NOTE**: When Igor passes a message to you, you *must* get the message, using GetXOPMessage, and get all of the arguments, using GetXOPItem, *before* doing any callbacks. The reason for this is that the act of doing the callback overwrites the message and arguments that Igor is passing to you.

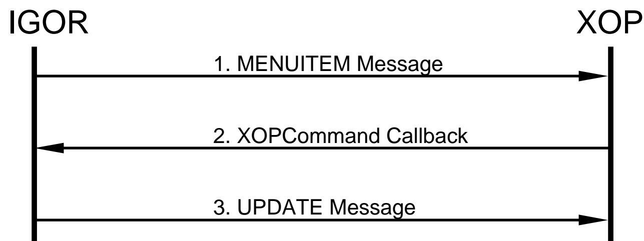
Some messages from Igor require that you return a result. Depending on the particular message, the result may be an error code, a pointer or a handle. You return the result by calling SetXOPResult. This XOPSupport routine stores the result in the **result** field of the IORecHandle.

When you do a callback to Igor, the XOPSupport routines may also use the result field to pass a result from Igor back to your XOP. For this reason, when you are returning a result to Igor, you must call SetXOPResult after doing any callbacks to Igor.

# Handling Recursion

This section describes some problems that can occur due to recursion and how to handle them. Simple XOPs will never encounter these problems. XOPs with windows or other elaborate XOPs are more likely to encounter them.

There are some circumstances in which Igor will send a message to your XOP while it is servicing a callback from your XOP. This is recursion and is illustrated in this diagram:




In this example, Igor sends the MENUITEM message to your XOP telling it that the user has invoked one of its menu items. As part of your XOP’s response, it does an XOPCommand callback to get Igor to do something. As a side effect, the XOPCommand callback causes Igor to update any windows that need updating. If your XOP has a window and it needs updating, Igor will send your XOP the UPDATE message while it is still working on the MENUITEM message.

Igor sends the UPDATE message to Macintosh XOPs only. Windows XOPs receive the WM_PAINT message from the Windows OS instead. However, the problem is the same - you receive a message at a possibly unexpected time, while you are in the middle of doing something else.

This problem can happen when you call the XOPCommand, XOPSilentCommand, SpinProcess and DoUpdate XOPSupport routines. Each of these routines will cause Igor to send an UPDATE message to your Macintosh XOP and may cause Windows to send a WM_PAINT message to your Windows XOP..

Although this example involves the MENUITEM and UPDATE or WM_PAINT messages, the problem is more general. Any time you do a callback to Igor, there is a chance that this will trigger Igor to send another message to you.

Recursion such as this can cause problems unless you write your XOP to handle it. The problems occur if, in servicing the UPDATE message from Igor or the WM_PAINT message from Windows, your XOP clobbers information that it needs to finish servicing the MENUITEM message or if your XOP is in a delicate state when it receives the XOP message. Fortunately, it is possible to avoid this.

There are two things that you must do to handle recursion:

1. When Igor calls you with a message, get the message and all of its arguments before doing any callbacks to Igor.
2. Don't use global variables to store things that will change from message to message. including the message and its arguments.

It turns out that most XOPs meet these requirements without any conscious effort on your part.

The reason for the first requirement is that Igor stores its message and arguments in your XOP's IORecHandle. In the example above, when Igor sends the UPDATE message, it clobbers the message and arguments for the MENUITEM message. This is no problem if, as shown in the sample XOPs, you use GetXOPMessage and GetXOPItems before doing any callbacks to Igor and if you store their results in local variables.

The second requirement is easily met if you just avoid using global variables as much as possible. This is good programming practice anyhow for reasons of modularity and insulation of routines from other routines.

If you use XOPCommand or XOPSilentCommand as part of your initialization, you must be aware that you may receive the UPDATE message before your initialization is complete. It is also possible to receive the UPDATE message at other times when your XOP is in the process of changing its state and can not service the update normally.

The solution for this is to use a flag to defer the true update until your XOP is ready to handle it. You can do this as follows:

1. Set a flag when you enter your XOPEntry routine.
2. If your XOPEntry routine is called again, while the flag is set, short-circuit the message from Igor (details below) and return.
3. Clear the flag when doing a normal exit from your XOPEntry routine.

For an UPDATE message on Macintosh, the short-circuiting works as follows:

1. Clear the update event by calling BeginUpdate followed immediately by EndUpdate. This is necessary because, if you ignore the UPDATE message, the Macintosh toolbox will send the update event over and over to Igor and this will block all other events.
2. Set a flag that will cause you to redraw your window at the next convenient opportunity.

# Data Sharing

Igor and your XOP need to access each other's data. For example, your XOP may need to access and manipulate Igor waves, numeric variables, and string variables. You may need to pass a handle containing your XOP's preferences to Igor, to be stored in the Igor Preferences file.

Data can be shared by Igor and an XOP in one of four ways:

- Via values
- Via pointers to local data (on the stack)
- Via pointers to data in the heap
- Via handles to data in the heap

Values (e.g., ints, longs) and pointers are built-in elements of the C programming language. They are documented in C programming books. No further explanation is needed.

Handles are extensions to C. Igor uses Macintosh-style handles, even when running on Windows. The term "handle", as used in this manual, means a Macintosh-style handle (data type "Handle"). Windows-style handles (data type "HANDLE") are not used in any Igor-XOP interactions. Handles are used in XOP programming relating to waves, menus, and strings, as well as for other purposes.

When dealing with memory objects that are shared between Igor and your XOP, you must use Macintosh memory management routines. The XOP Toolkit provides emulation for the necessary Macintosh routines. When dealing with your own private data, you can use Macintosh routines, standard C memory management routines, or Windows memory management routines.

Since manipulating memory pervades programming, it is important that you understand how XOP memory management is done. If you are not familiar with Macintosh memory management, take a detour to the section **Macintosh Memory Management** on page 261.

When dealing with Igor objects, such as waves, you never allocate and dispose memory directly. Instead you call XOPSupport routines, such as MakeWave and KillWave. See Chapter 7 for details on accessing Igor data.

# Igor/XOP Compatibility Issues

As Igor evolves, WaveMetrics makes every attempt to maintain backward compatibility so that your XOP will continue to work with new versions of Igor. However, you still need to make sure that your XOP is compatible with the version of Igor that is running.

There are two situations that need to be handled:

- The running Igor is too old to work with your XOP.
- Your XOP is too old to work with the running Igor.

To avoid compatibility problems you must do two things:

- Check the version of Igor that is running.
- Set the fields of your XOPI resource correctly.

## Checking Igor's Version

It is very easy to make sure that the running Igor is not too old. The XOPInit routine in XOPSupport.c, which your XOP calls from its main function, sets a global variable, igorVersion, to the version number of the currently running Igor. For version 5.00, igorVersion will be 500.

For example, if your XOP uses features added to Igor in version 5.00, you must restrict your XOP to running with Igor Pro 5.00 or later. Put the following in your XOP's main function:

```
if (igorVersion < 500) {        // Requires Igor Pro 5.00 or later.
   SetXOPResult(OLD_IGOR);
   return;
}
```

This test must be placed after your call to XOPInit and SetXOPEntry.

OLD_IGOR is an XOP custom error code, defined in the XOP's .h file, with a corresponding error string in the XOP's .r or .rc file. The error string should be something like "VDT2 requires Igor Pro 5.00 or later" so that the user knows what version of Igor is required. XOP custom errors are discussed on page 128.

If you want your XOP to work with older versions of Igor then you must check which version of Igor you are running with before using features added in newer versions. Here is a trivial example.

```
static int
MakeTextWave(char* name, long numPoints)
{
   waveHndl wavH;
   long dimensionSizes[MAX_DIMENSIONS+1];

   if (igorVersion < 300)     // Text waves were added in Igor Pro 3.0
      return IGOR_OBSOLETE;

   MemClear(dimensionSizes, sizeof(dimensionSizes));
   dimensionSizes[0] = numPoints;
   return MDMakeWave(&wavH,name,NULL,dimensionSizes,TEXT_WAVE_TYPE,1);
}
```

The test on igorVersion is not really necessary in this case because MDMakeWave does its own testing and will return IGOR_OBSOLETE if the running version of Igor can not make the wave as requested. It is important, however, that the routine that calls this routine be prepared to handle an error.

If you do plan to support older versions of Igor, check the description of the XOPSupport routines in Chapter 13 to see which version of Igor Pro they require and how they deal with older versions.

## XOP Protocol Version

The first field of an XOP's XOPI resource stores the "XOP protocol version" and is set to the value XOP_VERSION. When Igor examines XOPs at launch time, it checks this field. If the value in the field is too low or too high, Igor concludes that the XOP is not compatible and ignores it. This provides a way for Igor to make sure that your XOP is sufficiently compatible that it is OK to call its main function where the XOP can do further version checking.

Igor 1.24 through 1.28 worked with XOPs whose XOP protocol version was 2 or 3.

Igor Pro 2 through Igor Pro 5 work with XOPs whose XOP protocol version is 2, 3 or 4. The current value of XOP_VERSION is 4.

Chapter 5

# Adding Operations


Overview ........ 146
Igor Pro 4 Compatibility ........ 147
The Sequence of Events ........ 147
The XOPC 1100 Resource ........ 148
Operation Categories ........ 149
Choose a Distinctive Operation Name ........ 150
Operation Handler ........ 151
Creating Starter Code ........ 152
Operation Parameters ........ 154
The Command Template ........ 155
Optional Parameters ........ 157
Mnemonic Names ........ 158
The Runtime Parameter Structure ........ 159
String Parameters ........ 163
IMPORTANT ........ 163
Name Parameters ........ 163
Wave Parameters ........ 164
Wave Range Parameters ........ 164
VarName Parameters ........ 165
DataFolderAndName Parameters ........ 166
Structure Parameters ........ 168
External Operation Structure Parameter Example ........ 169
Extended Structure Parameters ........ 172
Extended Structure Parameter Example ........ 173
Runtime Output Variables ........ 174
Starter Code Details ........ 175
Updating Starter Code ........ 176
Supporting Old XOPs ........ 177

Operation Handler Checklist ........................................................................ 178

# Overview

An external operation, like a built-in Igor operation, is a routine that performs an action but has no explicit return value. An operation has the following syntax:

```
<Operation Name> [optional flags] <parameter list>
```

The optional flags generally take one of the following forms:

| Form | Description |
|---|---|
| `/F` | A flag by itself. Turns an operation feature on. |
| `/F=f` | A flag with a numeric or string parameter. |
| `/F=(f1, f2, . . .)` | A flag with a set of similar numeric or string parameters. |
| `/F=[f1, f2, . . .]` | A flag with a set of numeric indices. |
| `/F={f1, f2, . . .}` | A flag with a list of disparate numeric or string parameters. |

If your XOP will run with Igor Pro 5 or later only you can use multi-character flags such as /ABCD. The limit is four characters.

<parameter list> consists of one or more numeric or string parameters or keywords. Here are some common forms:

| Form | Examples |
|---|---|
| Numeric expression | `3; sin(K0/3); 3 + sin(K0/3)` |
| String expression | `"Hello"; date(); "Today is" + date()` |
| Wave list | `wave0, root:wave1, :wave2` |
| Keyword=Value | `baud=9600, dayOfWeek=Monday, size={3.4,5.2}` |

A parameter must be separated from the next with a comma.

When you add an operation to Igor, you first need to decide the name of the operation and what flags and parameters it will have. Usually, it is possible to use a syntax similar to an existing Igor operation or external operation.

The main steps in creating an external operation are:

- Creating the XOPC resource which tells Igor what operations your XOP adds.
- Creating an operation template which tells Igor the syntax of your operation.
- Writing an ExecuteOperation function which implements the operation.
- Writing the code that registers the operation with Igor when your XOP is loaded.

## Igor Pro 4 Compatibility

This chapter explains how to create an external operation using Igor Pro 5 or later. Igor Pro 5 includes a feature called “Operation Handler” which greatly simplifies the process of creating an external operation. In addition, Operation Handler makes it possible to call external operations directly from Igor user functions. Prior to Igor Pro 5, to call an external operation from a user function you had to use a kludge involving Igor’s Execute operation.

If your XOP must run with Igor Pro 4, you must use the old method of implementing an external operation. To reduce clutter, the old method is not documented in this manual. See Chapter 5 in the XOP Toolkit 3.1 manual which is included on the XOP Toolkit CD ROM or contact WaveMetrics support to receive the XOP Toolkit 3.1 manual in PDF form.

## The Sequence of Events

When Igor starts up, it searches the Igor Extensions folder and subfolders, looking for XOP files. When it finds an XOP file, Igor looks for an XOPC 1100 resource in the file. This resource, if it exists, defines the operation or operations that the XOP adds to Igor.

If the user invokes one of these operations from Igor’s command line or from a macro, Igor loads the XOP into memory and calls its main function (if it’s not already loaded). The XOP’s main function calls RegisterOperation, an XOPSupport routine through which the XOP tells Igor the syntax of the operation and the address of the XOP function which Igor should call to implement it.

Next Igor parses the operation parameters and uses them to fill a structure. It then calls the XOP function, passing the structure to it.

The XOP carries out the operation and returns a result code to Igor. If the result code is non-zero, Igor displays an error dialog with an appropriate error message.

The sequence of events is similar when an external operation is called from a user-defined function except that Igor loads the XOP when the function is compiled in order to determine the operation’s syntax.

# The XOPC 1100 Resource

As an illustration, here is the XOPC resource for the SimpleLoadWave sample XOP.

```
// Macintosh, in the SimpleLoadWave.r file.
resource 'XOPC' (1100) {                    // Operations added by XOP.
   {
      "SimpleLoadWave",                     // Name of operation.
      XOPOp + UtilOP + compilableOp,        // Operation category specifier.
   }
};

// Windows, in the SimpleLoadWaveWinCustom.rc file.
1100 XOPC                                   // Operations added by XOP.
BEGIN
   "SimpleLoadWave\0",                      // Name of operation.
   XOPOp | utilOp | compilableOp,           // Operation category specifier.

   "\0"         // NOTE: NULL required to terminate the resource.
END
```

An XOP can add multiple operations to Igor. In this case, there will be additional name/category pairs in the XOPC resource.

The compilableOp flag tells Igor that this is an Operation Handler-compatible external operation. If this flag is missing, Igor will deal with the external operation using the old method documented in the XOP Toolkit 3.1 manual.

## Operation Categories

The operation category specifier is some combination of the symbols shown in the following table. The operation category controls when the operation will appear in Igor's Help Browser Command Help list. For example, if the category includes the XOPOp and dataOp bits, it will appear in the list when All, External or Wave Analysis are selected in the Operations popup menu.

| Symbol | Bit Value | Operation Help Dialog Category |
|---|---|---|
| displayOp | 1 | Other Windows |
| waveOp | 2 | About Waves |
| dataOp | 4 | Wave Analysis |
| cursorOp | 8 | Controls and Cursors |
| utilOp | 16 | Programming & Utilities |
| XOPOp | 32 | External |
| compilableOp | 64 | Operation Handler-compatible |
| graphOp | 128 | Graphs |
| tableOp | 256 | Tables |
| layoutOp | 512 | Layouts |
| allWinOp | 1024 | All Windows |
| drawOp | 2048 | Drawing |
| ioOp | 4096 | I/O |
| printOp | 8192 | Printing |

The compilableOp bit does not affect the Help Browser. It tells Igor that the external operation is implemented using Operation Handler.

For most XOPs the category will include the XOPOp and compilableOp bits plus at least one other bit, depending on the functionality of the operation. If in doubt, use utilOp.

# Choose a Distinctive Operation Name

**NOTE**: The names of an XOP's operations must not conflict with the present or future names of Igor's built-in operations or functions or with the present or future names of any other XOP's operations or functions. *Don't use a vague or general name.* If you choose a name that clearly describes the operation's action, chances are that it will not conflict.

The name that you choose for your operation becomes unavailable for use as a wave name, window name or global variable name. For example, if you name your operation Test then you can not use the name Test for any Igor object. If you have existing experiments that use Test, they will be in conflict with your XOP. Therefore, *pick a name that is unlikely to be useful for any other purpose*. This is especially important if you plan to share your XOP with other Igor users since they are probably not prepared to deal with errors when they open existing experiments.

## Operation Handler

Operation Handler is a part of Igor that makes it easy to create an external operation that can be executed from the command line, from a macro or from a user function. It was added in Igor Pro 5.

Command line and macro operations are interpreted while user function operations are compiled and then later executed. The Operation Handler simplifies implementation of an operation in all of these modes.

Here is an outline of what you need to do to use Operation Handler:

1. Define a command template - a string that describes the syntax and parameters of your operation.
2. Define a runtime parameter structure which contains fields for your operation's parameters arranged in the prescribed order.
3. Write a routine that registers your operation with Igor. This is called your RegisterOperation function.
4. Write a routine that executes your operation, given a pointer to your runtime parameter structure. This is called your ExecuteOperation function.
5. Set the compilableOp bit in the XOPC resource for the operation.
6. When your XOP is initialized, call your RegisterOperation function to register your operation.

The good news is that Operation Handler can do steps 2 and 3 for you and can get you started on step 4. This is described in detail under **Creating Starter Code** in the next section.

When your operation is invoked from the command line, from a macro or from a user function, Igor will process any supplied parameters and then call your ExecuteOperation function, passing the runtime parameter structure to you. You will use the parameters to perform the operation.

The Operation Handler handles all of the details of compiling your operation in a user function and of parsing parameters when your operation is invoked from the command line or from a macro.

Before we get into the details, we will take a short side trip to see how Operation Handler generates starter code for you.

# Creating Starter Code

Operation Handler can automatically generate starter code for you, including a complete definition of your runtime parameter structure, a mostly complete definition of your RegisterOperation function, and a skeleton for your ExecuteOperation function. To get your starter code, you execute a ParseOperationTemplate command from within Igor. ParseOperationTemplate takes your command template and generates your starter code and stores it in the clipboard. Anything previously in the clipboard is overwritten. You can then paste into your source file.

The best way to use this feature is to create an experiment to store the template for your external operation and create a function within the experiment to create the starter code. For an example, see the experiment "SimpleLoadWave Template.pxp". You can create your own template experiment by duplicating that file.

"SimpleLoadWave Template.pxp" contains the following code:

```
Menu "Macros"
   "CopySimpleLoadWaveCode"
End

Function CopySimpleLoadWaveCode()   // Copies starter code to the clipboard.
   String cmdTemplate = "SimpleLoadWave"
   cmdTemplate += " " + "/A[=name:ABaseName]"
   cmdTemplate += " " + "/D"
   cmdTemplate += " " + "/I"
   cmdTemplate += " " + "/N[=name:NBaseName]"
   cmdTemplate += " " + "/O"
   cmdTemplate += " " + "/P=name:pathName"
   cmdTemplate += " " + "/Q"
   cmdTemplate += " " + "/W"
   cmdTemplate += " " + "[string:fileParamStr]"

   ParseOperationTemplate/T/S=1/C=2 cmdTemplate
End
```

CopySimpleLoadWaveCode copies the starter code to the clipboard. After executing it (or your version of it), you can paste from the clipboard into your source file. The generated code includes the following:

A comment showing the operation template.

A complete runtime parameter structure.

A skeleton ExecuteOperation function that you need to fill out.

A mostly complete RegisterOperation function which you need to call from your main function.

Before proceeding, you may want to open the "SimpleLoadWave Template.pxp" experiment in Igor and execute the CopySimpleLoadWaveCode function. Then paste the generated code into a text file and take a look at it. Don't worry about the details at this point. They are explained in the following sections.

Igor Pro 5.00 and 5.01 had a bug in the automatic code generation for external operations. See page 541 for details.

# Operation Parameters

Parameters come in two main kinds: flags and main parameters. Main parameters come in two types: simple main parameters and keyword=value main parameters.

```
SampleOp /F=3/K={A,3.2}     wave0, wave1
            |                |       |
          flags           simple main parameters

SampleOp /F=3/K={A,3.2}     waves={wave0,wave1}
            |                |
          flags           keyword=value main parameter
```

Flags and keywords can specify a single value (/A=1) or a set of values (/A={1,2,3}). A flag and its values, a keyword and its values, or a simple parameter are each called a "parameter group". Each individual value associated with a flag, keyword or simple parameter is called a "parameter".

Your command template can specify any number of flag groups. As for main parameters, it can specify any number of keyword groups or simple groups, but you can not mix keyword and simple groups as main parameters in the same operation.

When the user invokes the operation, flags and keywords may appear in any order. Simple parameters, if they appear at all, must appear in the order specified in the command template.

Optional parameters are supported as described below on page 157. Parameters that are required (i.e., not optional) must be supplied when the operation is compiled or interpreted. If required parameters are missing, Operation Handler detects this and generates the appropriate error message.

## The Command Template

You must provide a command template when you call RegisterOperation. A command template is a plain text string. The format of the command template is:

```
<OperationName> <flag descriptions> <main parameter descriptions>
```

For example:

```
SampleOp /O /P=name /Y={number,number} key1={number,string}
```

This says that the operation name is SampleOp, that it accepts three flag groups and one keyword group. In this example, the parameter groups are:

```
   /O                           // A flag group with no parameters
   /P=name                      // A flag group with one parameter
   /Y={number,number}           // A flag group with two parameters
   keyword1={number,string}     // A keyword group with two parameters
```

The name of the operation must appear in the template without any leading spaces. A space is required before the start of the main parameters. Otherwise spaces are optional and can be added anywhere.

Flag and keyword parameter groups can use parentheses, brackets or braces to enclose a list of parameters. Usually braces are used. The leading parenthesis, bracket or brace is called the "prefix" character and the trailing parenthesis, bracket or brace is called the "postfix" character.

Parameters within a group are separated by commas.

In flag parameters, there is no separator between one group and the next:

```
SampleOp /A=number /B=string
```

In main parameters, one group is separated from the next by a comma:

```
SampleOp number, string
SampleOp key1=number, key2=string
```

There is one exception to this. The keyword 'as' can be used in place of comma between one simple main parameter and the next. This is a special case that supports syntax like this:

```
Save wave as string
```

In this case, ‘as’ would appear both in the command template and in the actual command.

This special case is intended to support save-file syntax. It does not work if the first parameter is numeric.

The recognized parameter type keywords are:

```
number
string
name
wave
waveRange
varName
dataFolderAndName
structure (Igor Pro 5.03 or later)
```

Most operations will use number, string, name and wave parameters. The varName, waveRange, dataFolderAndName and structure types are less common.

The waveRange parameter type is used to allow the user to specify an input wave or a subset of a 1D input wave, for example wave1(x1,x2) or wave1[p1,p2]. Igor's CurveFit operation is an example of an operation that supports this syntax.

The varName parameter type is used when a parameter must be the name of a local or global variable or NVAR or SVAR. This applies when an operation wants to return a value via a variable whose name is specified as a parameter. For example, Igor's Open operation takes a refNum parameter which is the name of a numeric variable, and stores a file reference number in the specified variable.

The dataFolderAndName parameter type is typically used for operations that create waves. For example, Igor's Duplicate operation takes a destWave parameter. Since the destination wave may not exist when Duplicate runs, the destWave parameter can not be of type wave. Using a dataFolderAndName parameter allows the user to reference an object that does not yet exist.

The structure parameter type would be used by advanced programmers to pass pointers to structures between an Igor user-defined function and an XOP.

# Optional Parameters

Optional parameters are designated using brackets. There are three types of optional parameters, as the following examples illustrate:

```
// Optional flag or keyword parameters
SampleOp /A[=<parameters>]
SampleOp key1[=<parameters>]

// Normal optional parameters
SampleOp /A={number[,string, wave]}
SampleOp key1={number[,string, wave]}
SampleOp number[,string, wave]

// Array-style optional parameters
SampleOp /A={number, string[3]}
SampleOp key1={number, string[3]}
SampleOp number, string[3]
```

Using the optional flag or keyword syntax allows the user to supply or omit the equals sign and subsequent parameters after a flag or keyword.

Using the normal optional parameter syntax allows you to designate that some parameters are required and the rest are optional. Do not nest sets of brackets for this purpose. Just one set is allowed.

Using the array-style optional parameter syntax allows you to specify that zero or more parameters of a particular type appear at the end of a parameter group or at the end of the main parameter list.

When creating a Save Wave type of operation, you can combine the 'as' keyword and the array-style optional parameters like this:

```
Save wave[100] as string
```

In this case at least one wave must be specified in the command before the ‘as’ keyword. Support for this syntax was added in Igor Pro 5.02. If you use this syntax, you must check Igor’s version as described on page 140.

Other than the usages shown above, no other use of brackets to indicate optional parameters is allowed. For example, the following is not needed or allowed:

```
SampleOp [ /A/B=number/C=string ]       // WRONG
```

All flag and keyword groups are always optional. In other words, the user can always omit any flag or keyword and this fact is not to be indicated by brackets in the template.

As explained below, you can do a runtime check to see if a particular parameter group was supplied when the operation was invoked. You can also do a runtime check to see if a parameter within a group was supplied.

When the user invokes the operation, simple main parameters must appear in the order given by the command template if they appear at all. Flag groups and keyword groups can appear in any order.

If a flag or keyword group takes a prefix character (parenthesis, bracket, brace) and all of the parameters except the first are optional, Operation Handler will allow the user to invoke the group without the prefix character and with just one parameter. This allows you to change the syntax of a group from:

```
/A=value or keyword=value
```

to

```
/A=={value1[, value2]} or keyword={value1[, value2]}
```

and yet to maintain backward compatibility. Old Igor procedure code that omits the prefix character will still work and new code that uses the prefix character will also work. At execution time, you test to see if the second parameter was set.

# Mnemonic Names

Although it is not required, you can and should include mnemonic names in your template. Here is an example template that includes mnemonic names.

```
SampleOp /O /P=name:pathName /Y={number:offset, number:scale}
```

The underlined words are mnemonic names. At present the only use for mnemonic names is to allow Igor to generate more meaningful starter code for you. In the future, Igor may use the mnemonics for other purposes and may require the mnemonic name. Consequently, it is strongly recommended that you supply them.

## The Runtime Parameter Structure

When you create your starter code, Operation Handler will define a structure into which it will store parameters and other values at runtime. When your operation is invoked, Igor will pass this structure to you. The format of the runtime parameter structure is best understood through a simple example. Assume that your command template is:

```
SampleOp /A=number:aNum /B={string:bStrH,name:bName} /C=wave:cWaveH
           key1={number:key1Num,string:key2StrH},
           key2={name:key2Name,wave:key2WaveH}
```

When you create your starter code, Igor will define this runtime parameter structure:

```
#include "XOPStructureAlignmentTwoByte.h" // Set structure alignment.

struct SampleOpRuntimeParams {
   // Flag parameters.

   // Parameters for /A flag group.
   int AFlagEncountered;
   double aNum;
   int AFlagParamsSet[1];

   // Parameters for /B flag group.
   int BFlagEncountered;
   Handle bStrH;
   char bName[MAX_OBJ_NAME+1];
   int BFlagParamsSet[2];

   // Parameters for /C flag group.
   int CFlagEncountered;
   waveHndl cWaveH;
   int CFlagParamsSet[1];

   // Main parameters.

   // Parameters for key1 keyword group.
   int key1Encountered;
   double key1Num;
   Handle key1StrH;
   int key1ParamsSet[2];

   // Parameters for key2 keyword group.
   int key2Encountered;
   char key2Name[MAX_OBJ_NAME+1];
   waveHndl key2WaveH;
   int key2ParamsSet[2];

   // These are postamble fields that Igor sets.
   int calledFromFunction;       // 1 if called from a user function.
   int calledFromMacro;          // 1 if called from a macro.
};
typedef struct SampleOpRuntimeParams SampleOpRuntimeParams;
typedef struct SampleOpRuntimeParams* SampleOpRuntimeParamsPtr;

#include "XOPStructureAlignmentReset.h"      // Reset structure alignment.
```

You can use any name for any field in the structure so long as the names are unique.

It is critical that the structures fields fit the specification and match the command template that you pass to RegisterOperation. If you change the command template and fail to change the structure or vice versa, a crash will likely occur.

All structures passed between Igor and an XOP use two-byte packing. You must use the #includes shown above to guarantee this.

You can use the calledFromFunction and calledFromMacro fields to determine how your operation was called, although this is usually not necessary.

For each parameter group, there will be a field to indicate if the group was encountered in the command, followed by a field for each parameter in the group, followed by an array that indicates which parameters in the group actually appeared in the command. In the example above, the BFlagEncountered will be non-zero if the command included a /B flag. The bStrH and bName fields contain the values specified for the parameters to the /B flag. The BParamsSet array contains an element for each parameter in the group and tells you if the corresponding parameter was present in the command.

For each number parameter, there is a corresponding double field, for each string parameter a corresponding Handle field, for each name or VarName parameter a corresponding array of MAX_OBJ_NAME+1 characters, for each wave parameter a corresponding waveHndl field, for each waveRange parameter a corresponding WaveRange field, for each dataFolderAndName parameter a corresponding DataFolderAndName field, and for each structure parameter a corresponding pointer field.

These parameter fields are arranged in groups that correspond to parameter groups.

In the example above, BParamsSet[0] tells you if the string parameter to /B was specified and BParamsSet[1] tells you if the name parameter to /B was specified. In this example, all of the parameters in the group are required, so, if BFlagEncountered is non-zero then BParamsSet[0] and BParamsSet[1] are guaranteed to also be non-zero and you don't need to test them. You would test the BParamsSet array elements if the /B flag had optional parameters. The number of elements in the ParamSet array must match the total number of parameters in the parameter group, including any optional parameters.

If your command template includes a flag or keyword with no parameters then there will be just one field for that group - the field that tells you if the flag or keyword was encountered in the command.

When used in a flag or keyword group, a set of array-style optional parameters is represented as an array in the structure. For example, if the command template contains

```
key3={string:key3StrH,number[2]:key3Num}
```

this is represented in the runtime parameter structure as:

```
int key3Encountered;
Handle key3StrH;
double key3Num[2];                  // Optional parameter.
int key3ParamsSet[3];
```

When used as a simple main parameter, a set of array-style optional parameters is also represented as an array in the structure. For example, if the command template ends with

```
string:strH, number[2]:num
```

this is represented in the runtime parameter structure as:

```
// Parameters for simple main group #0.
int strHEncountered;
Handle strH;
int strHParamsSet[1];

// Parameters for simple main group #1.
int numEncountered;
double num[2];                   // Optional parameter.
int numParamsSet[2];
```

At execution time, Operation Handler sets the "Encountered" field for each parameter group to sequential values starting from 1, indicating the order in which parameter groups were encountered. For most operations this order is immaterial and all you care about is if the field is zero or non-zero.

Each parameter group can be set only once by a single command. Operation Handler returns an error if a user tries to set a given parameter group twice.

## String Parameters

Each string parameter is passed to you in a handle. The handle for a string parameter can be NULL. This would happen if, for example, the user used an SVAR reference to pass a global string to you and the global string did not exist at runtime. You must always test a string handle. If it is NULL, do not use it. If the string is required for the operation, return the USING_NULL_STRVAR error code.

**IMPORTANT**

> Do not dispose string parameter handles. Also do not access string handles after your ExecuteOperation function returns. Igor will dispose them automatically when your ExecuteOperation function returns.

If you want to retain string data for later use, you must make your own copy of the string handle, using the HandToHand function or copy the string to a C string using GetCStringFromHandle. In this regard, external operations and external functions work differently. In an external operation, you must **not** dispose string parameter handles. In an external function, you must dispose them.

String handles are not C strings and are not null-terminated. Use GetHandleSize to determine the number of characters in the string. You can use the GetCStringFromHandle XOPSupport routine to move the characters into a C string.

## Name Parameters

Igor names consist of MAX_OBJ_NAME characters. An example of a name parameter is the name of an Igor symbolic path in a /P=pathName flag. The user can specify a name parameter as $"", in which case the corresponding name field of the runtime parameter structure will be an empty string. Usually, an operation treats this the same as if the parameter were not specified in the command.

## Wave Parameters

Waves are passed to you as wave handles. Wave handles always belong to Igor. You must never dispose or directly modify a wave handle.

The handle for a wave parameter can be NULL. This would happen if, for example, the user used a WAVE reference to pass a wave to you and the wave did not exist at runtime. You must always test a wave handle. If it is NULL, do not use it. If the wave is required for the operation, return a NOWAV error code.

The user can use * in place of a wave name when a wave parameter is expected. This will result in a null wave handle in the runtime parameter structure. If the wave is required for the operation, return a NOWAV error code. Otherwise, interpret this to mean that the user wants default behavior.

You must make sure that you can handle the data type of the wave. For example, if your operation requires a numeric wave, you must return an error if passed a text wave. Also check the numeric type and dimensionality if appropriate.

## Wave Range Parameters

Wave range parameters are passed to you as WaveRange structures. This structure is defined in IgorXOP.h:

```
struct WaveRange {
   waveHndl waveH;
   double startCoord;           // Start point number or x value
   double endCoord;             // End point number or x value
   int rangeSpecified;          // 1 if user specified range.
   int isPoint;                 // 0: X values. 1: Points.
};
```

The waveH field can be NULL. If it is, you should return NOWAV, unless you want to allow a null wave. As with the wave parameter, the user can use * in place of the wave name. Also as with the wave parameter, you must check the wave's type to make sure it is a type you can handle.

If the rangeSpecified field is zero, then the command did not specify a range of the wave. In this case, the isPoint field will be non-zero, the startCoord field will contain zero and the endCoord field will contain the number of the last point in the wave, treating the wave as 1D regardless of its actual dimensionality.

If the rangeSpecified field is non-zero then the command did specify a range. If isPoint is non-zero then startCoord and endCoord will contain point numbers. If isPoint is zero then startCoord and endCoord will contain X values.

Regardless of how the command was specified, you can use the CalcWaveRange routine to find the point numbers of the range of interest like this. This example assumes that the runtime parameter structure contains a WaveRange field named source.

```
long startPoint, endPoint;
int direction;

startPoint = p->source.startCoord;
endPoint = p->source.endCoord;
direction = 1;
if (p->source.rangeSpecified) {
   WaveRangeRec wr;

   MemClear(&wr,sizeof(WaveRangeRec));
   wr.x1 = p->source.startCoord;
   wr.x2 = p->source.endCoord;
   wr.rangeMode = 3;
   wr.isBracket = p->source.isPoint;
   wr.gotRange = 1;
   wr.waveHandle = p->source.waveH;
   wr.minPoints = 2;
   if (err = CalcWaveRange(&wr))
      return err;

   startPoint = wr.p1;
   endPoint = wr.p2;
   direction = wr.wasBackwards ? -1:1;
}
```

## VarName Parameters

A varName parameter is used in rare situations when the parameter to an operation is the name of a numeric or string variable into which the operation is to store a value. When invoking an operation with a varName parameter, the user can pass the name of a global variable, the name of a local variable, or, when executing from a user function, the name of an NVAR or SVAR.

When called from the command line or from a macro, the VarName field contains an actual variable name. When called from a user function, it actually contains binary data that Igor uses to locate the local variable, NVAR or SVAR into which data is to be stored. Consequently, you should never use the value of the VarName field, except to pass it to Igor.

To store a value in a variable referenced by a VarName parameter, you must use the StoreNumericDataUsingVarName or StoreStringDataUsingVarName XOPSupport functions. These call back to Igor which knows how to store into global variables, local variables, NVARs and SVARs.

# DataFolderAndName Parameters

A dataFolderAndName parameter is used when you need to get the name and location of a wave that may not yet exist. For example, the Duplicate operation takes a source wave and a destination wave name. The destination wave may or may not already exists.

A dataFolderAndName parameter can also be used to get the name of an existing data folder or the name of a data folder to be created.

The runtime parameter structure contains a DataFolderAndName structure for a corresponding dataFolderAndName parameter. The DataFolderAndName structure is defined in IgorXOP.h:

```
struct DataFolderAndName {
   DataFolderHandle dfH;
   char name[MAX_OBJ_NAME+1];
};
```

Typically the dfH and name fields of this structure would be passed to MDMakeWave to create a destination wave.

The DataFolderAndName type parameter is often used to allow the user to specify the name of the operation's destination wave. Traditionally, when an operation that allows you to specify a destination wave is compiled into a user function, if the user uses a simple name for the destination wave, the Igor compiler automatically creates a wave reference in the function for that wave. For example, if you write this:

```
Duplicate wave0, wave1
```

the Igor compiler automatically creates a wave reference for wave1, as if you wrote:

```
Duplicate wave0, wave1
Wave wave1
```

This automatic local wave reference is created only if the user uses a simple name, not if the user uses $<name>, a partial data folder path or a full data folder path.

The DataFolderAndName type parameter allows you to do the same thing, but you have to use special syntax for the mnemonic name when writing the operation template. Consider this operation template:

```
SampleOp DataFolderAndName:{dest,real}
```

This template declares a DataFolderAndName parameter with a mnemonic name "dest" which, when compiled into a user function, automatically creates a wave reference for the destination wave, if the user uses a simple name. The "real" keyword specifies the type of the destination wave. Other options are "complex" and "text".

As of Igor Pro 5.04, if a wave reference for the specified destination already exists when the operation is compiled, it will not attempt to create a new wave reference. This allows the Igor programmer to indicate the actual type of the destination wave for those operations, such as FFT, in which the destination wave can be of different types depending on operation parameters.

See the documentation for Igor's DWT operation for an example of how to document this automatic creation of wave references.

If your operation can create a destination wave of different types depending on circumstances, you should pick the most likely type. See the documentation for Igor's FFT operation for an example of how to document this behavior.

Using the special syntax shown above causes Igor to create an automatic wave reference under the conditions explained above. However, the automatic wave reference will be NULL until you set it by calling SetOperationWaveRef.

The SetOperationWaveRef callback sets the automatically created wave reference to refer to a specific wave, namely the wave that you created in response to the DataFolderAndName parameter. You must call it after successfully creating the destination wave.

SetOperationWaveRef will do nothing if no automatic wave reference exists.

Your code should look something like this:

```
// In your RuntimeParams structure
DataFolderAndName dest;
int destParamsSet[1];

// In your ExecuteOperation function
destWaveH = NULL;
err=MDMakeWave(&destWaveH,p->dest.name,p->dest.dfH,dimSizes,type,overwrite);
if (destWaveH != NULL) {
   int waveRefIndentifier = p->destParamsSet[0];
   err = SetOperationWaveRef(destWaveH, waveRefIndentifier);
}
```

SetOperationWaveRef was added in XOP Toolkit 5.04 and Igor Pro 5.04B05. If you call it with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

You can also use a DataFolderAndName parameter to get the name of a data folder rather than a wave. In this case, the dfH field of the DataFolderAndName structure will contain the parent data folder handle and the name field will contain the name of the child data folder which may or may not exist. The root data folder is a special case. If the root is specified in the command then the dfH field will contain the root data folder handle and the name field will be empty. Here is code that gets a handle for an existing data folder and takes the special case into account:

```
dataFolderH = p->df.dfH;
if (dataFolderH == NULL)
   return EXPECT_DATAFOLDER_NAME;
if (p->df.name[0] != 0) {
   if (err = GetNamedDataFolder(dataFolderH, p->df.name, &dataFolderH))
      return err;
}
```

## Structure Parameters

Igor Pro 5.03 added the ability to pass a pointer to a structure as a parameter to an external operation. This is a technique for advanced programmers.

If you use this feature, your XOP will require Igor Pro 5.03 or later. You should put a test in your main function to make sure that you are running with a recent enough version. See Checking Igor’s Version on page 140 for details.

Structure parameters are passed as pointers to structures. These pointers always belong to Igor. You must never dispose or resize a structure pointer but you may read and write its fields.

An instance of an Igor structure can be created only in a user-defined function and exists only while that function is running. Therefore, when a structure must be passed to an external operation, the operation must be called from a user-defined function, not from the command line or from a macro. An external operation that has an optional structure parameter can be called from the command line or from a macro if the optional structure parameter is not used.

The pointer for a structure parameter can be NULL. This would happen if the user supplies * as the parameter or in the event of an internal error in Igor. Therefore you must always test a structure parameter to make sure it is non-NULL before using it.

If you receive a NULL structure pointer as a parameter and the structure is required for the operation, return an EXPECTED_STRUCT error code. Otherwise, interpret this to mean that the user wants default behavior.

Here is an example of a command template that specifies a structure parameter:

```
DemoStructOp structure:{sp:DemoStruct}
```

In this example, “structure” is the parameter type, “sp” is the parameter name and “DemoStruct” is the name of the type of structure expected. When Igor compiles a call to your operation, it will require that the structure passed as a parameter be of the specified type. Thus, users of this

operation are forced to declare a structure type named DemoStruct. Choose a structure type name that is distinct and unlikely to be used for any other purpose.

You must make sure that the definition of the structure in Igor matches the definition in the XOP. Otherwise a crash is likely to occur.

In very rare cases, you might want to define an operation that takes any type of structure. In such cases you should use the extended form of the structure parameter, described under **Extended Structure Parameters** on page 172.

## External Operation Structure Parameter Example

Here is Igor procedure code which passes a structure to an external operation.

```
Constant kDemoStructVersion = 1000

Structure DemoStruct       // Structure of parameter to DemoStructOp.
   uint32 version          // Structure version.
   double num
   String str
EndStructure

Function TestDemoStructOp()
   String tmp

   STRUCT DemoStruct s

   s.version = kDemoStructVersion
   s.num = 1
   s.str = "Testing structure parameter"

   DemoStructOp s
End
```

Here is C code that implements the external operation. Most of this code would be automatically generated by Operation Handler when you execute this command within Igor:

```
ParseOperationTemplate/T/S=1/C=2 "DemoStructOp structure:{sp:DemoStruct}"

#include "XOPStructureAlignmentTwoByte.h"
#define kDemoStructVersion 1000        // 1000 means 1.000.
struct DemoStruct {
   unsigned long version;              // Structure version.
   double num;
   Handle strH;
};
typedef struct DemoStruct DemoStruct;
#include "XOPStructureAlignmentReset.h"
```

```
// Operation template: DemoStructOp structure:sp

// Runtime param structure for DemoStructOp operation.
#include "XOPStructureAlignmentTwoByte.h"
struct DemoStructOpRuntimeParams {
   // Parameters for simple main group #0.
   int spEncountered;
   DemoStruct* sp;
   int spParamsSet[1];

   // These are postable fields that Igor sets.
   int calledFromFunction;
   int calledFromMacro;
};
typedef struct DemoStructOpRuntimeParams DemoStructOpRuntimeParams;
typedef struct DemoStructOpRuntimeParams* DemoStructOpRuntimeParamsPtr;
#include "XOPStructureAlignmentReset.h"

static int
ExecuteDemoStructOp(DemoStructOpRuntimeParamsPtr p)
{
   DemoStruct* sp;
   char buf[256];
   char str[128];
   int err = 0;

   sp = NULL;

   // Flag parameters.

   if (p->spEncountered)
      sp = p->sp;

   if (sp == NULL) {
      strcpy(buf, "sp is NULL"CR_STR);
   }
   else {
      if (sp->version != kDemoStructVersion) {
         err = INCOMPATIBLE_STRUCT_VERSION;
         goto done;
      }
      if (err = GetCStringFromHandle(sp->strH, str, sizeof(str)))
         goto done;
      sprintf(buf, "sp->num = %g, sp->strH = \"%s\"", sp->num, str);
   }
   XOPNotice(buf);

done:
   return err;
}
```

```
int
RegisterDemoStructOp(void)
{
   char* cmdTemplate;
   char* runtimeNumVarList;
   char* runtimeStrVarList;

   cmdTemplate = "DemoStructOp structure:{sp:DemoStruct}";
   runtimeNumVarList = "";
   runtimeStrVarList = "";
   return RegisterOperation(cmdTemplate, runtimeNumVarList,
               runtimeStrVarList, sizeof(DemoStructOpRuntimeParams),
               (void*)ExecuteDemoStructOp, 0);
}
```

The Igor procedure code must declare a DemoStruct structure and the C code must declare a matching C structure. The use of the version field is a convention for preventing a crash if the procedure version of the structure and the C version get out-of-sync.

The Igor code declares the DemoStruct structure. It then creates and initializes an instance of it, named s. It then passes s (actually a pointer to s) to the DemoStructOp external operation.

The DemoStructOp external operation first checks that a valid pointer has been passed. It then checks that the version of the structure is compatible before accessing its fields. It can read, write or read and write any of the fields.

An advanced programmer may want to use a more flexible versioning scheme by allocating reserved fields in the structure and then using the version field to handle whatever version of the structure was passed from the Igor procedure. If possible you should simplify your life by using the simplified versioning technique shown above. However, if you decide to save structure data to disk and later read it back, you will have to deal with more complex versioning issues, as well as cross-platform byte-order issues. You will also have to deal with versioning issues if your XOP will be used by many people and you can not force them to update their Igor procedure code when you update your XOP.

For further important details see **Using Igor Structures as Parameters** on page 281.

## Extended Structure Parameters

In Igor Pro 5.04B07, an extended form of the structure parameter was added to allow you to determine the size and structure type name of the structure passed to the XOP. The main reason for adding this was to make it possible to create an operation that could accept any type of structure and write its contents to a file. This requires that we know the size of the structure.

The extended form of the template is:

```
DemoStructOp structure:{s,DemoStruct,1}
```

Here s is the mnemonic name for a structure parameter, DemoStruct is the parameter's structure type name, and 1 signifies that you want this to be an extended structure parameter.

When this form is used, the runtime parameter structure passed to the your operation's Execute routine contains a field of type IgorStructInfo instead of a pointer to the actual parameter. The IgorStructInfo structure contains fields that indicate the size of the structure parameter, the structure type name and a pointer to the structure parameter itself.

You should use the extended form if you want the added robustness provided by comparing the size and type name of the structure parameter passed to you to the expected size and type name. Another reason is to accept a structure parameter of any type, for example so that you can write it to a file.

If the user supplies * as the parameter for an extended form structure, the structSize field of the IgorStructInfo structure will be zero and the structPtr field will be NULL. You must test for this.

## Extended Structure Parameter Example

The code to implement the extended form structure parameter is the same as shown above except for the differences listed here.

The command template is slightly different:

```
ParseOperationTemplate/T/S=1/C=2 "DemoStructOp structure:{s:DemoStruct,1}"
```

The runtime parameter structure will have an IgorStructInfo field:

```
struct DemoStructOpRuntimeParams {
   // Parameters for simple main group #0.
   int sEncountered;
   IgorStructInfo s;    // Contains info about the structure parameter
   int sParamsSet[1];
```

The Execute operation will use the fields of the IgorInfoStruct:

```
static int
ExecuteDemoStructOp(DemoStructOpRuntimeParamsPtr p)
{
   IgorStructInfo* isip;
   DemoStruct* sp;
   char buf[256];
   char str[128];
   int err = 0;

   sp = NULL;

   // Flag parameters.

   if (p->sEncountered) {
      isip = &p->s;                 // Point to IgorStructInfo field.
      if (isip->structSize > 0) {   // 0 means user passed * for param.
         if (isip->structSize = sizeof(DemoStruct)) {
            err = OH_BAD_STRUCT_SIZE;
            goto done;
         }

         if (CmpStr(isip->structTypeName,"DemoStruct") != 0)) {
            err = OH_BAD_STRUCT_TYPE_NAME;
            goto done;
         }

         sp = p->sp;
      }
   }

   . . .    // The rest is the same as the previous example.
```

In this case there is no point to testing the structTypeName field since we specified that only structures of type DemoStruct could be passed to our operation. The structTypeName field may be of use in cases where any type of structure can be passed as the parameter.

# Runtime Output Variables

Some operations create numeric and/or string variables to pass information back to the calling routine. For example, the CurveFit operation creates V_chisq and V_Pr, among others. When an operation is called from a user function, if runtime lookup of globals is on (rtGlobals=1 or rtGlobals=2), the operation creates local variables. If runtime lookup of globals is off, it creates global variables. When invoked from macros, it creates local variables. When invoked from the command line, it creates global variables.

If you want to create runtime output numeric variables, you need to pass a semicolon-separated list of variable names as the runtimeNumVarList parameter to RegisterOperation. Numeric variable names must be legal standard Igor names and must begin with "V_". When your operation executes, you must call SetOperationNumVar to store a value in the variable. SetOperationNumVar takes care of determining if a local or global variable is being set.

Creating a runtime output string variable is the same except that you use the runtimeStrVarList parameter to RegisterOperation, string variable names must start with "S_", and you use SetOperationStrVar to store a value in the variable.

If you are updating a file loader XOP that calls SetFileLoaderOutputVariables, you need to change it to call SetOperationFileLoaderOutputVariables instead. You also need to specify that your operation sets V_flag, S_fileName, S_path and S_waveNames when you call RegisterOperation. For example, the RegisterOperation function for SimpleLoadWave contains these statements:

```
runtimeNumVarList = "V_flag;";
runtimeStrVarList = "S_path;S_fileName;S_waveNames;";
```

Operations should not use the value of variables as inputs. All inputs should be specified through operation parameters. There is no provision for an operation to test the value of a local variable at runtime.

# Starter Code Details

As explained on page 152, the starter code that Operation Handler generates for you includes the following:

A comment showing the operation template.

A complete runtime parameter structure.

A skeleton ExecuteOperation function that you need to fill out.

A mostly complete RegisterOperation function which you need to call from your main function.

When filling out the ExecuteOperation function, remember that you need to test string and wave handle parameters to make sure they are not null. It is a good idea to examine a completed ExecuteOperation function from a sample WaveMetrics XOP to see how NULL handles are handled.

The RegisterOperation function is complete except that you may need to add runtime numeric or string output variables as described on page 174.

For a more elaborate example which generates templates for multiple external operations, see "VDT2 Templates.pxp".

In the unlikely event that your command template exceeds 2048 characters, you will get an error in Visual C++. This is because of a line length limitation in Visual C++. You will need to break your command template up. See the NIGIPB2:NI488.c file for an example.

# Updating Starter Code

If you change your operation syntax, for example to add a new parameter, you must regenerate your starter code and transfer the new code to your source file. If you have started to fill out your ExecuteOperation function, you can not merely overwrite your old starter code. You need to preserve any additions you have made. Here is the recommended way to do this:

1. Open your template experiment and add the new parameter to your function.
2. Run your function to regenerate the starter code.
3. In your development system, paste the new starter code into a new window.
4. Copy the template comment and the runtime parameter structure, which you presumably have not modified, in their entirety from the new starter code and paste into your source file, overwriting the corresponding old code.
5. Copy the RegisterOperation from the new starter code and paste into your source file, overwriting the old code, except that, if you have specified runtime output variables, you must preserve those statements.
6. Find the section of the new ExecuteOperation function that deals with the new parameter and add that section to your old ExecuteOperation function.

# Supporting Old XOPs

For an old (pre-Operation Handler) XOP to be directly callable from an Igor user function, you need to rewrite it using Operation Handler techniques. This means that your XOP will require Igor Pro 5 or later. Also, using Operation Handler may require that you change the syntax of your operation because it does not support all possible variations of syntax.

The recommended approach is to freeze your old XOP and create a new version of it using Operation Handler. If your operation's syntax has to change, use a new name for the new version (e.g., VDT and VDT2). Users of your XOP will then have the option of using the old or new versions and you will be free to use Igor Pro 5 features.

When converting from the old method to Operation Handler, remove any calls like this from your code:

```
   SetXOPType(TRANSIENT);
```

XOPs that use Operation Handler must remain in memory once loaded.

If you update an XOP from the old way to Operation Handler, you may get compile errors in functions that use the operation. For example, imagine that you convert this Igor Pro 4 function:

```
Function Test()
   Execute "XLLoadWave" // Creates V_flag in global context
   NVAR V_flag
   Print V_flag
End
```

to this Igor Pro 5 function:

```
Function Test()
   XLLoadWave           // Call XLLoadWave directly in Igor Pro 5
   NVAR V_flag          // Compiler will complain about inconsistent type
   Print V_flag
End
```

The Igor Pro 5 compiler will complain. This is because XLLoadWave, when compiled, creates a local variable named V_flag. Then the NVAR statement attempts to create a reference to a global using the same name. The solution is to change the new function to:

```
Function Test()
   XLLoadWave  // Call XLLoadWave directly in Igor Pro 5
   Print V_flag
End
```

# Operation Handler Checklist

If your operation crashes or otherwise behaves unexpectedly, here are some things to double-check:

- You have set the compilableOp bit in the XOPC resource for your operation. If you have multiple operations, this bit must be set for each operation for which you want to use the Operation Handler.
- If your operation will be called directly, you do not have a call to SetXOPType(TRANSIENT) in your code.
- Your runtime parameter structure must be consistent with your command template. Reread the section entitled **The Runtime Parameter Structure** on page 159. See **Creating Starter Code** on page 152 and **Updating Starter Code** on page 176.
- You must always check string and wave handles to make sure they are not null before using them.
- You must not dispose string handles passed from Igor via your runtime parameter list. You also must not access such handles after your ExecuteOperation returns. If you want to keep the string data for later use, make a copy using HandToHand or GetCStringFromHandle.
- String handles are not C strings. Use GetHandleSize to determine the number of characters in the string. Use the GetCStringFromHandle XOPSupport routine to move the characters into a C string.
- You must never dispose or directly modify a wave handle. Wave handles belong to Igor.
- If you are updating a file loader XOP that calls SetFileLoaderOutputVariables, you need to change it to call SetOperationFileLoaderOutputVariables instead. You also need to specify that your operation sets V_flag, S_fileName, S_path and S_waveNames when you call RegisterOperation.

For other debugging ideas, see Chapter 12.

*Chapter* 6

# Adding Functions

Overview ........ 181
External Function Examples ........ 182
Adding an External Function to Igor ........ 182
The XOPF 1100 Resource ........ 183
    Function Categories ........ 185
Choose a Distinctive Function Name ........ 186
Invoking an External Function ........ 186
External Function Parameters and Results ........ 187
Parameter and Result Types ........ 188
Complex Parameters and Results ........ 189
Strings Parameters and Results ........ 190
Structure Parameters ........ 194
    External Function Structure Parameter Example ........ 196
Pass-By-Reference Parameters ........ 198
Keep External Functions in Memory ........ 201
FUNCTION Message Versus Direct Methods ........ 201
Error Checking and Reporting ........ 202

# Overview

An XOP can add any number of external functions as well as any number of external operations to Igor Pro.

An external function takes zero or more numeric, wave, string and structure parameters and returns a numeric or string result. An external function is similar to a user-defined function except that the external function is implemented by C or C++ code in an XOP whereas the user-defined function is implemented by Igor code in Igor's procedure window.

There are two reasons for creating external functions rather than user-defined functions. First, in some cases they may be significantly faster than user-defined functions. Second, you can do things, like sample an I/O port, from an external function that you can't do from a user-defined function.

Whereas an external operation has a variable number of parameters, an external function has a fixed number of parameters. Igor parses these parameters automatically and then passes them to the external function in a structure.

External functions can be called anywhere Igor can call a built-in function, including from a user-defined function, from a macro, from the command line, or in a curve-fitting operation.

To add a function to Igor, you must create an XOPF resource to specify the name of the function, the number and type of its parameters, and the type of its result. Then you need to write the code that implements the function. The implementation accesses the parameters through the structure that Igor passes to it. It returns two kinds of results. One is the function result, returned to the calling user-defined function. The other is an error code, which Igor uses to determine when a fatal error has occurred.

## External Function Examples

The XFUNC1 sample XOP, shipped with the XOP Toolkit, is a very simple XOP that adds three trivial external functions to Igor. The functions are XFUNC1Add(p1, p2), XFUNC1Div(p1, p2) and XFUNC1ComplexConjugate(p1). You can use this as a starting point for your external function XOP.

The XFUNC2 sample XOP implements two non-trivial external functions: logfit and plgndr.

logfit(wave, x) takes a single or double-precision wave containing three coefficients, a, b, and c, and an X value and returns a + b*log(c*x). Its use is illustrated by the XFUNC2 Demo.pxp example experiment shipped in the XFUNC2 folder.

plgndr(l, m, x) takes three numbers and returns the value of the Legendre polynomial at x. The use of plgndr is also illustrated in the XFUNC2 Demo.pxp experiment. The Legendre polynomial and its parameters, l and m, are described in *Numerical Recipes in C*.

The XFUNC3 sample XOP implements two trivial external string functions, xstrcat0(str1, str2) and xstrcat1(str1, str2). Both simply return the concatenation of string expressions str1 and str2. The XFUNC3 Demo.pxp experiment contains tests that measure the speed of these functions.

## Adding an External Function to Igor

When Igor Pro starts up, it examines each XOP file in the Igor Extensions folder or in its subfolders. It looks for an XOPF 1100 resource. This resource, if it exists, defines the functions that the XOP adds to Igor. Igor remembers the names of the functions added by the XOP.

If the XOP has an XOPF 1100 resource, Igor loads the XOP into memory at Igor's startup time. It calls the XOP's main function, telling the XOP to initialize itself. The XOP stays in memory from launch time until Igor quits.

# The XOPF 1100 Resource

For each external function added by an XOP, Igor needs to know the name of the function, the type of the function's return value and the number and type of parameters that the function expects. These things are defined in the XOPF 1100 resource which is defined in the XOP's .r file (*Macintosh*) or .rc file (*Windows*). For example, here is the XOPF resource for XFUNC1:

```
resource 'XOPF' (1100) {  // Macintosh, in XFUNC1.r.
    {
        "XFUNC1Add",                        // Function name
        F_UTIL | F_EXTERNAL,                // Function category
        NT_FP64,                            // Return type is double precision
        {
            NT_FP64,                        // Parameter types
            NT_FP64,
        },

        "XFUNC1Div",                        // Function name
        F_UTIL | F_EXTERNAL,                // Function category
        NT_FP64,                            // Return type is double precision
        {
            NT_FP64,                        // Parameter types
            NT_FP64,
        },

        "XFUNC1ComplexConjugate",           // Function name
        F_CMPLX | F_EXTERNAL,               // Function category
        NT_FP64 | NT_CMPLX,                 // Return type is double complex
        {
            NT_FP64 | NT_CMPLX,             // Double complex parameter
        },
    }
};

1100 XOPF                 // Windows, in XFUNC1WinCustom.rc.
BEGIN
    "XFUNC1Add\0",                          // Function name
    F_UTIL | F_EXTERNAL,                    // Function category
    NT_FP64,                                // Return type is double precision
        NT_FP64,                            // Parameter types
        NT_FP64,
        0,                // 0 terminates list of parameter types.

    "XFUNC1Div\0",                          // Function name
    F_UTIL | F_EXTERNAL,                    // Function category
    NT_FP64,                                // Return type is double precision
        NT_FP64,                            // Parameter types
        NT_FP64,
        0,                // 0 terminates list of parameter types.

    "XFUNC1ComplexConjugate\0",             // Function name
    F_CMPLX | F_EXTERNAL,                   // Function category
    NT_FP64 | NT_CMPLX,                     // Return type is double complex
        NT_FP64 | NT_CMPLX,                 // Double complex parameter
        0,                // 0 terminates list of parameter types.

    0,                    // 0 terminates the resource.
END
```

Here are the symbols available to specify the type of the external function's parameters and its return value.

| Symbol | Decimal | Where Used | Description |
|---|---|---|---|
| NT_FP64 | 4 | Return type, parameter type. | Double-precision floating point. |
| NT_FP64 \| NT_CMPLX | 5 | Return type, parameter type. | Complex. |
| HSTRING_TYPE | 8192 | Return type, parameter type. | String handle. |
| WAVE_TYPE | 16384 | Parameter type only. | Wave handle. |
| FV_REF_TYPE | 4096 | Parameter type only. | Indicates pass-by-reference parameter. Supported by Igor Pro 5 or later. |
| FV_STRUCT_TYPE | 1024 | Parameter type only. | Structure. Always use with FV_REF_TYPE. |

Note that other Igor number types, such as NT_FP32 and NT_I32, are not allowed. A parameter type can be WAVE_TYPE but not a return type. If a parameter is a wave, the XOPF resource does not specify what kind of wave it is (floating point numeric, integer numeric, text). Igor will allow the user to pass any kind of wave to your external function so you must check the wave type at run time. See the logfit routine in XFUNC2Routines.c for an example.

There is one case in which you do need to use a numeric type in conjunction with a wave type - when you are creating a curve fitting function. You must specify the first parameter type as WAVE_TYPE | NT_FP64. This is explained on page 188 in the section **Parameter and Result Types**.

The FV_REF_TYPE flag is used in conjunction with one of the other parameter types to signify that a parameter is passed by reference. It is used with numeric or string parameters and has no meaning for wave parameters. Pass-by-reference is described on page 198.

The ability to pass a structure to an external function was added in Igor Pro 5.03. FV_STRUCT_TYPE must always be ORed with FV_REF_TYPE.

## Function Categories

The function category specifier is some combination of the symbols shown in the following table. The function category controls when the function will appear in Igor's Help Browser. For example, specifying the function's category as F_EXTERNAL | F_SPEC would make the function appear when All, External or Special are selected in the Functions popup menu in the Command Help tab of the browser.

| Symbol | Bit Value | Function Help Dialog Category |
|---|---|---|
| F_TRIG | 1 | Trig |
| F_EXP | 2 | Exponential |
| F_SPEC | 4 | Special |
| F_CMPLX | 8 | Complex |
| F_TIMEDATE | 16 | Time and Date |
| F_ROUND | 32 | Rounding |
| F_CONV | 64 | Conversion |
| F_WAVE | 128 | About Waves |
| F_UTIL | 256 | Programming & Utility |
| F_NUMB | 512 | Numbers |
| F_ANLYZWAVES | 1024 | Wave Analysis |
| F_IO | 2048 | I/O |
| F_WINDOWS | 4096 | Windows |
| F_EXTERNAL | 8192 | External |
| F_STR | 32768 | String |

You must always set the F_EXTERNAL bit for all external functions.

## Choose a Distinctive Function Name

**NOTE**: The name of an external function must not conflict with the names of present or future built-in operations or functions or with the names of present or future external operations or functions. *Don't use a vague or general name.* If you choose a name that clearly describes the function's purpose, chances are that it will not conflict.

The name that you choose for your function becomes unavailable for use as a wave name, window name or global variable name. For example, if you name your function Test then you can not use the name Test for any Igor object. If you have existing experiments that use Test, they will be in conflict with your function. Therefore, *pick a name that is unlikely to be useful for any other purpose*. This is especially important if you plan to share your XOP with other Igor users since they are probably not prepared to deal with errors when they open existing experiments.

## Invoking an External Function

You can invoke an external function from Igor's command line, from a macro or from a user-defined function just as you would invoke a built-in function or user-defined function. For example, the XFUNC1Div function, defined by the XOPF 1100 resource of the XFUNC1 XOP, can be invoked from Igor's command line using the following commands:

```
XFUNC1Div(3,4)
Print XFUNC1Div(3,4)
K0 = XFUNC1Div(K1, K2)
wave0 = XFUNC1Div(wave1, wave2)
```

The last example is a wave assignment statement. You may think that the waves wave1 and wave2 are passed to the XFUNC1Div function. This is not the case. Instead, Igor calls XFUNC1Div one time for each point in wave0, passing a single value from wave1 and another single value from wave2 each time. This is because the parameter types for XFUNC1Div are NT_FP64. If the parameter types were WAVE_TYPE, Igor would pass waves. The logfit function implemented in XFUNC2 illustrates this.

# External Function Parameters and Results

The XFUNC1Div routine in XFUNC1.c illustrates how you access the function's parameters and how you return the result:

```
#include "XOPStructureAlignmentTwoByte.h"

struct XFUNC1DivParams  {
   double p2;              // p2 is the second parameter to XFUNC1Div.
   double p1;              // p1 is the first parameter to XFUNC1Div.
   double result;
};
typedef struct XFUNC1DivParams XFUNC1DivParams;

#include "XOPStructureAlignmentReset.h"

static int
XFUNC1Div(XFUNC1DivParams* p) // p is a pointer passed from Igor to XFUNC.
{
   p->result = p->p1 / p->p2; // XFUNC result.

   return 0;                  // XFUNC error code
}
```

The #include statements insure that Igor and the XOP agree on the alignment of the parameter structure. See **Structure Alignment** on page 279 for details.

The parameter passed from Igor to XFUNC1Div is p. p is a pointer to a structure. The structure contains the parameters that the external function is to operate on. Parameters are passed by Igor to the external function in reverse order. The first element in the structure is the last parameter to the XFUNC1Div function. The last element in the structure is where the external function stores its result. This result field is the value that Igor returns to the calling function.

If an external function has a wave parameter, the corresponding field in the structure must be of type waveHndl. If an external function has a string parameter or result, the corresponding field must be of type Handle. If an external function has a structure parameter, the corresponding field must be of type pointer to structure. All other parameters are numbers and the corresponding fields must be declared double.

In this example, the parameters are passed by value. Numeric and string parameters can also be passed by reference, as described on page 198. In that case, the structure fields would be defined as double* or Handle*.

Note that the function returns an error code which is zero if no error occurred or a built-in Igor error code (defined in IgorXOP.h) or an XOP-defined error code (defined in the XOP's .h file with corresponding error strings in the STR# 1100 resource). Do not confuse this error code with the result returned to the calling function via the result field in the structure. In the case of

XFUNC1Div, there is nothing to go wrong so the error code is always zero. If you return a non-zero error code, Igor will abort procedure execution and display a dialog indicating the nature of the error. You should do this only in the case of fatal errors.

# Parameter and Result Types

The XFUNC2 XOP adds an external function, logfit, that takes a wave and a numeric value as its parameters and returns a numeric value. The XOPF 1100 resource for this XOP looks like this:

```
resource 'XOPF' (1100) {   // Macintosh, in XFUNC2.r.
   {
      // y = a + b*log(c*x)
      "logfit",                    // Function name
      F_EXP | F_EXTERNAL,          // Function category
      NT_FP64,                     // Return value type
      {
         WAVE_TYPE | NT_FP64,      // Wave
         NT_FP64,                  // Double-precision x
      },

      . . .       // Other function definitions here.
   }
}

1100 XOPF                  // Windows, in XFUNC2WinCustom.rc.
BEGIN
   // y = a + b*log(c*x)
   "logfit\0",                     // Function name
   F_EXP | F_EXTERNAL,             // Function category
   NT_FP64,                        // Return value type
      WAVE_TYPE | NT_FP64,         // Wave
      NT_FP64,                     // Double precision x
      0,          // 0 terminates list of parameter types.

   . . .          // Other function definitions here.

   0,             // NOTE: 0 required to terminate the resource.
END
```

This says that the logfit function returns a double-precision number (NT_FP64). WAVE_TYPE | NT_FP64 indicates that the first parameter to logfit is a wave. The second parameter is a double-precision number.

The logfit function is written to handle single-precision or double-precision wave parameters. It checks the type of the wave parameter and returns an error if the type is not one of these.

Normally, you can use just WAVE_TYPE for wave function parameters. However, if the function is a curve-fitting function, as in this example, you must use WAVE_TYPE | NT_FP64. The use of NT_FP64 here is needed to satisfy the FuncFit operation. However, it does not mean that Igor will pass only double-precision waves to the function. You must still check the wave type in the function.

**NOTE**: Igor can pass a NULL wave handle to your function. This happens during the execution of a user-defined function when a wave that is assumed to exist does not exist. Your function must check for this as shown in the logfit example.

# Complex Parameters and Results

XFUNC1 adds a function that returns the complex conjugate of a complex parameter. The XOPF 1100 resource for a complex function looks like this:

```
// Macintosh, in XFUNC1.r.
resource 'XOPF' (1100) {
   {
      "XFUNC1ComplexConjugate",         // Function name
      F_CMPLX | F_EXTERNAL,             // Function category
      NT_FP64 | NT_CMPLX,               // Return value type = DPC
      {
         NT_FP64 | NT_CMPLX,            // DPC parameter
      },
   }
};

// Windows, in XFUNC1WinCustom.rc.
1100 XOPF
BEGIN
   "XFUNC1ComplexConjugate\0",          // Function name
   F_CMPLX | F_EXTERNAL,                // Function category
   NT_FP64 | NT_CMPLX,                  // Return value type = DPC
      NT_FP64 | NT_CMPLX,               // DPC parameter
      0,          // 0 terminates list of parameter types.
   0,             // 0 terminates the resource.
END
```

The first use of NT_FP64 | NT_CMPLX specifies that the return value is double-precision, complex. The second use of NT_FP64 | NT_CMPLX specifies that the parameter is double-precision, complex.

See XFUNC1ComplexConjugate in XFUNC1.c for the code that implements a complex function.

# Strings Parameters and Results

The XFUNC3 XOP adds an external function that takes two string parameters and returns a string result. The XOPF 1100 resource for this XOP looks like this:

```
// Macintosh, in XFUNC3.r.
resource 'XOPF' (1100) {
   {
      // str1 = xstrcat0(str2, str3)
      "xstrcat0",                  // Function name
      F_STR | F_EXTERNAL,          // Function category (string)
      HSTRING_TYPE,                // Return value type is string handle.
      {
         HSTRING_TYPE,             // First parameter is string handle.
         HSTRING_TYPE,             // Second parameter is string handle.
      },

      // str1 = xstrcat1(str2, str3)
      "xstrcat1",                  // Function name
      F_STR | F_EXTERNAL,          // Function category (string)
      HSTRING_TYPE,                // Return value type is string handle.
      {
         HSTRING_TYPE,             // First parameter is string handle.
         HSTRING_TYPE,             // Second parameter is string handle.
      },
   }
};

// Windows, in XFUNC3WinCustom.rc.
1100 XOPF
BEGIN
   // str1 = xstrcat0(str2, str3)
   "xstrcat0\0",                   // Function name
   F_STR | F_EXTERNAL,             // Function category (string)
      HSTRING_TYPE,                // Return value type is string handle.
      HSTRING_TYPE,                // First parameter is string handle.
      HSTRING_TYPE,                // Second parameter is string handle.
      0,          // 0 terminates list of parameter types.

   // str1 = xstrcat1(str2, str3)
   "xstrcat1\0",                   // Function name
   F_STR | F_EXTERNAL,             // Function category (string)
      HSTRING_TYPE,                // Return value type is string handle.
      HSTRING_TYPE,                // First parameter is string handle.
      HSTRING_TYPE,                // Second parameter is string handle.
      0,          // 0 terminates list of parameter types.

   0,             // 0 terminates the resource.
END
```

This resource defines two string functions that do exactly the same thing. The function declarations are identical except for the function name. The reason for having identical functions is to demonstrate two methods by which Igor can call an external function. These methods are explained on page 201 under **FUNCTION Message Versus Direct Methods**.

The return value type of HSTRING_TYPE says that the function returns a string. The HSTRING_TYPEs indicate both parameters are strings.

This function is invoked as follows:

```
String aString
aString = xstrcat0("Hello", " out there")
```

xstrcat is defined as follows:

```
static int
xstrcat(                        // str1 = xstrcat(str2, str3)
   struct {
      Handle str3;
      Handle str2;
      Handle result;
   }* p)
{
   Handle str1;                 // output handle
   long len2, len3;
   int err=0;

   str1 = NULL;                 // If error occurs, result is NULL.
   if (p->str2 == NULL) {       // Error -- input string does not exist.
      err = NO_INPUT_STRING;
      goto done;
   }
   if (p->str3 == NULL) {       // Error -- input string does not exist.
      err = NO_INPUT_STRING;
      goto done;
   }

   len2 = GetHandleSize(p->str2);   // length of string 2
   len3 = GetHandleSize(p->str3);   // length of string 3
   str1 = NewHandle(len2 + len3);   // Get output handle.
   if (str1 == NULL) {
      err = NOMEM;
      goto done;  // out of memory
   }

   memcpy(*str1, *p->str2, len2);
   memcpy(*str1+len2, *p->str3, len3);

done:
   if (p->str2 != NULL)
      DisposeHandle(p->str2); // Get rid of input parameters.
   if (p->str3 != NULL)
      DisposeHandle(p->str3); // Get rid of input parameters.
   p->result = str1;

   return err;
}
```

The error code NO_INPUT_STRING is defined by the XOP whereas NOMEM is a standard Igor error code (defined in IgorXOP.h).

Strings in Igor are stored in plain Macintosh-style handles, even when running on Windows. The handle contains the string's text, with neither a count byte nor a trailing null byte. Use GetHandleSize to find the number of characters in the string. To use C string functions on this text you need to copy it to a local buffer and null-terminate (using GetCStringFromHandle) it or add a null terminator to the handle and lock the handle. If you pass the handle back to Igor, you must remove the null terminator and unlock the handle.

An external string function returns a handle as its result. It can return NULL in the event of an error.

Unlike a wave handle, a handle passed as a string parameter to an external function belongs to that function. The external function must dispose of the handle. Alternately, the external function can return the handle as the function result after possibly modifying its contents.

**NOTE**: As for wave handles, it is possible for a string parameter to be passed as NULL because the string does not exist during the execution of a compiled user-defined function. The external function must check for this.

xstrcat checks its parameters. If either is NULL, it returns NULL as a result. Otherwise, it creates a new handle and concatenates its input parameters into the new handle. This new handle is stored as the result and the input handles are disposed. The result handle belongs to Igor. The external function must not dispose of it or access it in any way once it returns to Igor.

## Structure Parameters

Igor Pro 5.03 added the ability to pass a pointer to a structure as a parameter to an external function. This is a technique for advanced programmers.

If you use this feature, your XOP will require Igor Pro 5.03 or later. You should put a test in your main function to make sure that you are running with a recent enough version. See **Checking Igor's Version** on page 140 for details.

Structure parameters are passed as pointers to structures. These pointers always belong to Igor. You must never dispose or resize a structure pointer but you may read and write its fields.

An instance of an Igor structure can be created only in a user-defined function and exists only while that function is running. Therefore, external functions that have structure parameters can be called only from user-defined functions, not from the command line or from macros.

The pointer for a structure parameter can be NULL. This would happen if the user supplies * as the parameter or in the event of an internal error in Igor. Therefore you must always test a structure parameter to make sure it is non-NULL before using it.

If you receive a NULL structure pointer as a parameter and the structure is required for the operation, return an EXPECTED_STRUCT error code. Otherwise, interpret this to mean that the user wants default behavior.

Unlike the case of external operations, the Igor compiler has no way to know what type of structure your external function expects. Therefore, Igor will allow any type of structure to be passed to an external function. When writing Igor procedures that call the external function, you must be careful to pass the right kind of structure.

You must make sure that the definition of the structure in Igor matches the definition in the XOP. Otherwise a crash is likely to occur.

Here is the XOPF resource declaration for an external function that takes one parameter which is a pointer to a structure:

```
// Macintosh
resource 'XOPF' (1100) {
   {
      // result = XtestF1Struct(struct F1Struct* s)
      "XTestF1Struct",              // Function name
      F_EXTERNAL,                   // Function category
      NT_FP64,                      // Return value type
      {
         FV_STRUCT_TYPE | FV_REF_TYPE, // Pointer to F1Struct.
      },
   }
};

// Windows
1100 XOPF
BEGIN
   // result = XTestF1Struct(struct F1Struct* s)
   "XTestF1Struct\0",               // Function name
   F_EXTERNAL,                      // Function category
   NT_FP64,                         // Return value type
      FV_STRUCT_TYPE | FV_REF_TYPE, // Pointer to F1Struct
      0,                            // 0 terminates parameters.

   0,                               // 0 terminates functions.
END
```

The use of FV_STRUCT_TYPE | FV_REF_TYPE says that the function takes a pointer to a structure as a parameter.

## External Function Structure Parameter Example

Here is Igor procedure code which invokes this external function.

```
Constant kF1StructVersion = 1000    // 1000 means 1.000

Structure F1Struct         // Structure of parameter to XTestF1Struct.
   uint32 version          // Structure version.
   double num
   String strH
   Wave waveH
   double out
EndStructure

Function Test()
   STRUCT F1Struct s

   s.version = kF1StructVersion
   s.num = 4321
   s.strH = "This is a test."
   Wave s.waveH = jack
   s.out = 0

   Variable result = XTestF1Struct(s)
End
```

Here is C code that implements the external function:

```
#include "XOPStructureAlignmentTwoByte.h" // Set structure alignment.

#define kF1StructureVersion 1000          // 1000 means 1.000.
struct F1Struct {                         // Structure of parameter.
   unsigned long version;                 // Structure version.
   double num;
   Handle strH;
   waveHndl waveH;
   double out;
};
typedef struct F1Struct F1Struct;

struct F1Param {                          // Parameter structure.
   F1Struct* sp;                          // This param is pointer to struct.
   double result;
};
typedef struct F1Param F1Param;

#include "XOPStructureAlignmentReset.h"
```

```
int
XTestF1Struct(struct F1Param* p)
{
   struct F1Struct* sp;
   char buffer[256];
   char str[128];
   char nameOfWave[MAX_OBJ_NAME+1];
   int err=0;

   sp = p->sp;
   if (sp == NULL)  {
      err = EXPECT_STRUCT;
      goto done;
   }

   if (sp->version != kF1StructureVersion ) {
      err = INCOMPATIBLE_STRUCT_VERSION;
      goto done;
   }

   sp->out = 1234;

   if (sp->strH == NULL) {
      err = USING_NULL_STRVAR;          // Error: input string does not exist
      goto done;
   }

   if (sp->waveH == NULL) {
      err = NOWAV;                      // Error: expected wave
      goto done;
   }

   if (err = GetCStringFromHandle(sp->strH, str, sizeof(str)-1))
      goto done;                        // String too long.

   WaveName(sp->waveH, nameOfWave);

   sprintf(buffer, "num=%g, str=\"%s\", wave='%s'"CR_STR,
                              sp->num, sp->str, nameOfWave);
   XOPNotice(buffer);

done:
   p->result = err;
   return err;
}
```

There are two structures involved in this example: the F1Param structure and the F1Struct structure. F1Param defines the format of the *parameter structure* – the structure through which

Igor passes any and all parameters to the external function. F1Struct defines the format of the one *structure parameter* passed to this function. p is a pointer to a parameter structure. sp is a pointer to a particular parameter which is a pointer to an Igor structure whose format is defined by F1Struct.

The Igor procedure code must declare an F1Struct structure and the C code must declare a matching C structure. The use of the version field is a convention for preventing a crash if the procedure version of the structure and the C version get out-of-sync.

The Igor code declares the F1Struct structure. It then creates and initializes an instance of it, named s. It then passes s (actually a pointer to s) to the XTestF1Struct external function.

The XTestF1Struct external function first checks that a valid pointer has been passed. It then checks that the version of the structure is compatible before accessing its fields. It can read, write or read and write any of the fields.

An advanced programmer may want to use a more flexible versioning scheme by allocating reserved fields in the structure and then using the version field to handle whatever version of the structure was passed from the Igor procedure. If possible you should simplify your life by using the simplified versioning technique shown above. However, if you decide to save structure data to disk and later read it back, you will have to deal with more complex versioning issues, as well as cross-platform byte-order issues. You will also have to deal with versioning issues if your XOP will be used by many people and you can not force them to update their Igor procedure code when you update your XOP.

For further important details see **Using Igor Structures as Parameters** on page 281.

# Pass-By-Reference Parameters

External functions can have pass-by-reference parameters. Only numeric and string parameters can be pass-by-reference.

The use of pass-by-reference XFUNC parameters requires Igor Pro 5 or later. If your function must run with Igor Pro 4 then you must not define it to use pass-by-reference.

Normally function parameters are pass-by-value. This means that the called function (the external function in this case) can not modify the value of the parameter as seen by the calling user-defined function.

With a pass-by-reference parameter, the called function **can** modify the parameter's value and the calling user-defined function will receive the modified value on return. Pass-by-reference allows a function to return multiple values to the calling function, in addition to the function result.

There are three differences in the implementation of a pass-by-reference parameter compared to a normal pass-by-value parameter.

First, in the XOPF resource, when declaring the parameter type, you use the FV_REF_TYPE flag. Here is an example of an XOPF that declares a function with three pass-by-reference parameters:

```
resource 'XOPF' (1100) {
   {
      // number = ExampleFunc(number,complexNumber,str)
      "ExampleFunc",                  // Function name
      F_UTIL | F_EXTERNAL,            // Function category
      NT_FP64,                        // Return value type
      {                               // Parameter types
         NT_FP64 | FV_REF_TYPE,
         NT_FP64 | NT_CMPLX | FV_REF_TYPE,
         HSTRING_TYPE | FV_REF_TYPE,
      },
   }
};
```

Second, in the parameter structure passed to the XFUNC, the fields corresponding to pass-by-reference parameters must be pointers to doubles or pointers to Handles. For example:

```
// All structures passed between Igor and XOP are two-byte aligned.
#include "XOPStructureAlignmentTwoByte.h"

struct ExampleParams {             // Fields are in reverse order.
   Handle* strHPtr;                // Parameter is pass-by-reference.
   double* complexNumberPtr;       // Parameter is pass-by-reference.
   double* scalarNumberPtr;        // Parameter is pass-by-reference.
   double result;                  // Function result field.
};
typedef struct ExampleParams ExampleParams;
typedef ExampleParams* ExampleParamsPtr;

// Reset structure alignment to default.
#include "XOPStructureAlignmentReset.h"
```

The third difference has to do with string handles. As with pass-by-value, when your XFUNC receives a pass-by-reference string parameter, you own the handle pointed to by the Handle* field, strHPtr in this case. The difference is that you must also return a handle via this field and Igor owns the handle that you return. Often it is convenient to reuse the input handle as the output handle, as the example code below shows, in which case ownership of the handle passes from the

XOP back to Igor and you must not dispose it. If you return a different handle then you must dispose the input handle.

As with pass-by-value, strings passed by reference can be NULL and you must check for this.

This example shows how to access and modify pass-by-reference parameters:

```
int
ExampleFunc(ExampleParamsPtr p)
{
   char str[256];
   int err;

   err = 0;

   if (*p->scalarNumberPtr != 0) {
      char buffer[512];

      // GetCStringFromHandle tolerates NULL handles.
      GetCStringFromHandle(*p->strHPtr, str, sizeof(str)-1);
      sprintf(buffer, "ExampleFunc received: %g, (%g,%g), %s"CR_STR,
         *p->scalarNumberPtr,
         p->complexNumberPtr[0], p->complexNumberPtr[1],
         str);
      XOPNotice(buffer);
   }

   *p->scalarNumberPtr *= 2;
   p->complexNumberPtr[0] *= 2;
   p->complexNumberPtr[1] *= 2;

   // *p->strHPtr will be NULL if caller passes uninitialized string.
   if (*p->strHPtr == NULL)
      *p->strHPtr = NewHandle(0L);

   if (*p->strHPtr != NULL) {
      strcpy(str, "Output from ExampleFunc");
      err = PutCStringInHandle(str, *p->strHPtr);
   }

   /* Do not dispose *p->strHPtr. Since it is a pass-by-reference parameter,
      the calling function retains ownership of this handle.
   */

   p->result = 0;

   return err;
}
```

## Keep External Functions in Memory

All XOPs with external functions must remain resident all of the time. You can satisfy the by just not calling the SetXOPType(TRANSIENT) XOPSupport routine.

## FUNCTION Message Versus Direct Methods

When an external function is invoked, Igor calls code in the XOP that defined the function. There are two methods for calling the code:

- The FUNCTION message method.
- The direct-call method.

Nearly all external functions should use the direct method which is faster than the FUNCTION message method. The only except is if your external function must run on Macintosh with Igor Pro 4 and if it directly or indirectly calls GetResource. In that case, you should use the FUNCTION message method. Otherwise your calls to GetResource might fail.

The FUNCTION message method of calling the external function uses the same techniques for communication between Igor and the XOP as all other XOP Toolkit messages. Igor calls your XOP's XOPEntry function, passing it the FUNCTION message. This message has two arguments. The first argument is an index number starting from zero that identifies which of your external functions is being invoked. For the XFUNC1 XOP, if the XFUNC1Add function is being invoked, the index is 0. If the XFUNC1Div function is being invoked, the index is 1. The second argument is a pointer to a structure containing the parameters to the function and a place to store the function result as described above.

The DoFunction routine in XFUNC1.c decides which of the external functions is being invoked. It then calls that function, passing it the necessary pointer. It acts as a dispatcher.

When calling a direct function, Igor does not send a FUNCTION message to your XOPEntry routine but instead calls your function code directly, passing to it a pointer to the structure containing the parameters and a place to store the function result.

Here is how Igor decides which method to use when calling an external function. When Igor starts up it examines each XOP's XOPF 1100 resource to see if it adds an external function or functions. If the XOP does add functions, Igor loads the XOP into memory and sends it the INIT message (i.e., calls the XOP's main function). Then it sends the XOP the FUNCADDRS message once for each external function that the XOP adds. The FUNCADDRS message has one argument: the index number for an external function added by the XOP. If the XOP returns NULL in response to this message then Igor will call that external function using the FUNCTION

message method. If the XOP returns other than NULL, this result is taken as the address of the routine in the XOP for Igor to call directly when the external function is invoked.

XFUNC3 illustrates both ways of registering a function. In response to the FUNCADDRESS message from Igor, it registers the xstrcat0 function as direct by returning an address, and registers the xstrcat1 function as message-based by returning NULL .

## Error Checking and Reporting

The logfit function in XFUNC2 illustrates error checking and reporting. The function is designed for use as a curve-fitting function but it also can be invoked from an Igor procedure as follows:

```
K0 = logfit(wave0, x);
```

The value that will be stored in K0 is returned via the result field of the structure that Igor passes to the logfit function. In addition to this result, the function has a return value which Igor looks at to see if the function encountered a serious problem.

logfit is defined as follows:

```
#include "XOPStructureAlignmentTwoByte.h"
struct LogFitParams {
   double x;                // Independent variable.
   waveHndl waveHandle;     // Coefficient wave (contains a, b, c coefs).
   double result;
};
typedef struct LogFitParams LogFitParams;
typedef struct LogFitParams *LogFitParamsPtr;
#include "XOPStructureAlignmentReset.h"

int
logfit(LogFitParamsPtr p)  // y = a + b*log(c*x)
{
   double* dPtr;           // Pointer to double-precision wave data
   float* fPtr;            // Pointer to single-precision wave data
   double a, b, c;

   // Check that wave handle is valid
   if (p->waveHandle == NULL) {
      SetNaN64(&p->result);      // Return NaN if wave not valid.
      return NON_EXISTENT_WAVE;
   }

   // Check coefficient wave's numeric type.
   switch (WaveType(p->waveHandle)) {
      case NT_FP32:
         fPtr = WaveData(p->waveHandle);
         a = fPtr[0];
         b = fPtr[1];
         c = fPtr[2];
         break;
      case NT_FP64:
         dPtr = WaveData(p->waveHandle);
         a = dPtr[0];
         b = dPtr[1];
         c = dPtr[2];
         break;
      default:                   // Can't handle this data type.
         p->result = nan(255);   // Return NaN.
         return REQUIRES_SP_OR_DP_WAVE;
   }

   p->result = a + b*log10(c*p->x);

   return 0;
}
```

The LogFitParams structure is defined in XFUNC2.h.

The error codes NON_EXISTENT_WAVE and REQUIRES_SP_OR_DP_WAVE are defined in XFUNC2.h with corresponding strings in XFUNC2's STR# 1100 resource.

Notice that logfit starts off by making sure that the wave parameter is not NULL. This is necessary because the external function can be called from an Igor user-defined function. User-defined functions are compiled by Igor and can refer to a wave that doesn't exist at compile time. If, when the compiled function is executed, the wave still does not exist, it will be passed to logfit as a NULL handle.

When Igor passes a wave to an external function, *it does not do type conversion*. The wave passed to logfit could be single precision, double precision or integer. It could also be complex. It could even be a text wave. Logfit returns NaN if the wave passed to it is other than single or double precision, real.

Returning a non-zero value from an external function causes Igor to abort procedure execution and display an error alert. Therefore, you should only do this if the error is fatal - that is, if it is likely to make any further processing meaningless. If we did not want passing the wrong type of wave to logfit to be a fatal error, we would set p->result to NaN, as shown above, and then return zero to Igor, instead of NON_EXISTENT_WAVE or REQUIRES_SP_OR_DP_WAVE.

Chapter 7

# Accessing Igor Data


Overview ........ 207
Waves ........ 207
Routines for Accessing Waves ........ 207
Getting a Handle to a Wave ........ 207
Getting Wave Properties ........ 208
Setting Wave Properties ........ 208
Locking and Unlocking Wave Handles ........ 209
Reading Wave Data ........ 209
Writing Wave Data ........ 209
Example ........ 210
Wave Data Types ........ 211
Accessing Numeric Wave Data ........ 212
The Point Access Method ........ 212
The Temporary Storage Access Method ........ 213
The Direct Access Method ........ 214
Speed Comparisons ........ 215
Organization of Numeric Data in Memory ........ 216
Accessing Text Wave Data ........ 217
Wave Scaling and Units ........ 218
Numeric and String Variables ........ 220
Routines for Accessing Variables ........ 221
Creating Variables ........ 221
Getting Variable Contents ........ 221
Setting Variable Contents ........ 222
Example ........ 223
Dealing With Data Folders ........ 224
Routines for Accessing Data Folders ........ 224
Getting a Handle to a Data Folder ........ 225

Creating and Killing a Data Folder ..................................................... 225
Accessing Objects in a Data Folder .................................................... 226
Data Folder Conventions ........................................................................... 227

# Overview

This chapter explains how to access Igor data – waves, numeric variables, string variables and the data folders that contain them.

Dealing with waves, string variables, and data folders involves the use of Macintosh-style handles, whether your XOP is running on Macintosh or Windows. You need to understand Macintosh-style handles in order to access and manipulate Igor data. See **Data Sharing** on page 139 and **Macintosh Memory Management** on page 261 for details.

# Waves

A wave is stored in a block of memory in the heap, referenced by a handle. The first hundred or so bytes of the block contains a structure that describes the wave. After that comes the main wave data followed by optional wave data such as the wave note. You will often need to get a wave handle from Igor or pass a wave handle to Igor to identify a particular wave to operate on.

You never need to and never should attempt to deal with the wave structure directly. Instead, you should use the XOPSupport routines, provided in XOPWaveAccess.c and listed in Chapter 13, to get and set wave data and properties.

## Routines for Accessing Waves

The XOPSupport routines for dealing with waves are described in detail in Chapter 13. Here is a summary of the commonly-used routines.

### Getting a Handle to a Wave

| Routine | Description |
|---|---|
| MakeWave | Makes 1D waves. |
| MDMakeWave | Makes waves of dimension 1 through 4. |
| FetchWave | Returns a handle to a wave in the current data folder given its name. |
| GetDataFolderObject | Returns a handle to a wave given its name and data folder. |

In addition, you can create external functions and external operations that receive wave handles from Igor as a parameter.

### Getting Wave Properties

| Routine | Description |
|---|---|
| WaveName | Returns wave name given a handle. |
| WaveType | Returns data type. |
| WavePoints | Returns number of elements in all dimensions. |
| MDGetWaveDimensions | Returns number of elements in each dimension. |
| WaveScaling | Gets scaling for the row dimension only. |
| MDGetWaveScaling | Gets scaling for the specified dimension. |
| WaveUnits | Gets row dimension units, data units. |
| MDGetWaveUnits | Gets units for the specified dimension or data units. |
| MDGetDimensionLabel | Gets dimension label for specified dimension index. |
| GetWaveDimensionLabels | Gets all dimension labels for entire wave. |
| WaveNote | Gets the wave note. |
| WaveLock | Returns the lock state of the wave. |

### Setting Wave Properties

| Routine | Description |
|---|---|
| SetWaveScaling | Sets scaling for the row dimension only. |
| MDSetWaveScaling | Sets wave scaling for the specified dimension. |
| SetWaveUnits | Sets row dimension units, data units. |
| MDSetWaveUnits | Sets units for the specified dimension or data units. |
| MDSetDimensionLabel | Sets dimension label for specified dimension index. |
| SetWaveDimensionLabels | Sets all dimension labels for entire wave. |
| SetWaveNote | Sets the wave note. |
| SetWaveLock | Sets the lock state of the wave. |

### Locking and Unlocking Wave Handles

| Routine | Description |
|---|---|
| MoveLockHandle | Locks wave handle and returns previous state. You must restore the locked/unlocked state using HSetState. |
| GetWavesInfo | Locks multiple wave handles and returns pointers to their data. You must restore the locked/unlocked state using SetWaveStates. |
| SetWavesStates | Restores locked/unlocked state. Used with GetWavesInfo. |

### Reading Wave Data

| Routine | Description |
|---|---|
| WaveData | Returns pointer to the start of wave's data. |
| MDAccessNumericWaveData | Returns the offset to the wave data. |
| MDGetNumericWavePointValue | Returns the value of a single element of wave. |
| MDGetDPDataFromNumericWave | Returns all wave data in a double-precision buffer. |
| MDGetTextWavePointValue | Gets the value of a single element of text wave. |
| GetTextWaveData | Gets the entire contents of a text wave. |

### Writing Wave Data

| Routine | Description |
|---|---|
| WaveData | Returns pointer to the start of wave's data. |
| MDAccessNumericWaveData | Returns the offset to wave data. |
| MDSetNumericWavePointValue | Sets the value of a single element of wave. |
| MDStoreDPDataInNumericWave | Sets all wave data from a double-precision buffer. |
| MDSetTextWavePointValue | Sets the value of a single element of text wave. |
| SetTextWaveData | Sets the entire contents of a text wave. |

## Example

Here is a simple example that illustrates creating a wave, filling it with values and setting a wave property. The underlined items are functions, constants or structures defined in the XOPSupport library and headers.

```
static int
MakeWave0(void)
{
   waveHndl wavH;
   char waveName[MAX_OBJ_NAME+1];
   DataFolderHandle dfH;
   long dimensionSizes[MAX_DIMENSIONS+1];
   float* wp;
   long p;
   double offset, delta;
   int hState;
   int err;

   strcpy(waveName, "wave0");
   dfH = NULL;                     // Put wave in the current data folder
   MemClear(dimensionSizes, sizeof(dimensionSizes));
   dimensionSizes[ROWS] = 100;     // Make 1D wave with 100 points
   if (err = MDMakeWave(&wavH, waveName, dfH, dimensionSizes, NT_FP32, 1))
      return err;

   hState=MoveLockHandle(wavH);    // Lock wave handle in heap
   wp = WaveData(wavH);            // Get a pointer to the wave data
   for(p = 0; p < 100; p++)
      *wp++ = p;                   // Store point number in wave
   HSetState(wavH, hState);        // Restore lock/unlocked state

   offset = 0; delta = .001;       // Set X scaling to .001/point
   if (err = MDSetWaveScaling(wavH, ROWS, &delta, &offset))
      return err;

   return 0;
}
```

This routine makes a 100 point, single-precision (NT_FP32) 1D wave. The letters “MD” in some of the XOPSupport routine names indicate that the routine is capable of dealing with multi-dimensional as well as 1D waves.

We get a pointer to the main wave data using the XOPSupport WaveData routine. While we use the pointer to the wave data, we lock the wave in the heap so that the data can not move. In this example, the locking and unlocking is not necessary because we are doing nothing that could

cause the heap blocks to move. In general, it is necessary to lock the wave when you access its data through a pointer.

This code is capable of dealing with single-precision numeric data only. You will see below that waves can contain one of 8 floating point and integer numeric types, can be real or complex, and also can contain text. An XOP programmer can choose to support all of these types or just some of them. XOPSupport routines described below provide ways to access wave data of any type.

## Wave Data Types

Waves can have one of the following data types:

| Symbol | Decimal Value | Bytes Per Element | Description |
|---|---|---|---|
| NT_FP64 | 4 | 8 | Double-precision floating point (double). |
| NT_FP32 | 2 | 4 | Single-precision floating point (float). |
| NT_I32 | 32 | 4 | Signed 32-bit integer (long). |
| NT_I16 | 16 | 2 | Signed 16-bit integer (short). |
| NT_I8 | 8 | 1 | Signed 8-bit integer (char). |
| NT_I32 \| NT_UNSIGNED | 32+64 | 4 | Unsigned 32-bit integer (long). |
| NT_I16 \| NT_UNSIGNED | 16+64 | 2 | Unsigned 16-bit integer (short). |
| NT_I8 \| NT_UNSIGNED | 8+64 | 1 | Unsigned 8-bit integer (char). |
| TEXT_WAVE_TYPE | 0 | variable | Unsigned 8-bit integer (char). |

In addition, any of the types, except for TEXT_WAVE_TYPE, can be ORed with NT_CMPLX to specify a complex data type. For example, (NT_FP32 | NT_CMPLX) represents complex, single-precision floating point, has a decimal value of 2+1 and takes 2*4 bytes per element.

As illustrated in the following sections, you can write your XOP to handle any data type or to require specific data types.

## Accessing Numeric Wave Data

This section describes how Igor stores numeric wave data and how you can use the XOPSupport wave access routines to access the data. The routines mentioned here are described in detail in Chapter 13. The WaveAccess sample XOP illustrates how to use them.

The XOP Toolkit supports three different ways to access the numeric data in waves of dimension 1 through 4. The **point access** method is the easiest and the slowest. The **direct access** method is the hardest and the fastest. The **temporary storage** method is not too hard and reasonably fast. This section discusses how to choose which method to use.

### The Point Access Method

The easiest method is to use the MDGetNumericWavePointValue and MDSetNumericWave-PointValue routines to access a single wave point at a time. For example:

```
long indices[MAX_DIMENSIONS];
double value[2];     // real part and imag part (if wave is complex)
int result;

MemClear(indices, sizeof(indices));    // Clear unused dimensions.
indices[0] = row;
indices[1] = column;
if (result = MDGetNumericWavePointValue(wavH, indices, value))
   return result;
value[0] += 1;                         // Add one to real part
if (result = MDSetNumericWavePointValue(wavH, indices, value))
   return result;
```

This example assumes that the wave has two dimensions and changes a value in a given row and column.

This method is simple because you deal with indices, not with pointers, and you don't need to worry about the numeric type of the data. The XOPSupport routines do any numeric conversions that are needed. It will continue to work with future versions of Igor that support additional numeric types or use a different organization of data in memory. The downside of this method is that it is not as fast as possible. The reason for this is that, each time you call either of the routines, it must calculate the address of the wave data specified by the indices and then must convert the data into double precision floating point (unless it is already in double precision).

Unless your application deals with large arrays and/or accesses the same data points over and over and over again, the speed penalty will not be very noticeable and the point access method is a good one to choose. This is also the recommended method if you are not comfortable dealing with pointers.

When storing into an integer wave, MDSetNumericWavePointValue truncates the value that you are storing. If you want, you can do rounding before calling MDSetNumericWavePointValue.

### The Temporary Storage Access Method

In the temporary storage access method, you use MDGetDPDataFromNumericWave and MDStoreDPDataInNumericWave to transfer data between the wave and a temporary storage buffer that you create. For example:

```
long numBytes;
double* dPtr;
double* dp;
int result;

numBytes = WavePoints(wavH) * sizeof(double);    // Bytes needed for copy
if (WaveType(wavH) & NT_CMPLX)                   // Complex data?
   numBytes *= 2;
dPtr = (double*)NewPtr(numBytes);
if (dPtr==NULL)
   return NOMEM;

if (result = MDGetDPDataFromNumericWave(wavH, dPtr)) {   // Get copy.
   DisposePtr((Ptr)dPtr);
   return result;
}
dp = dPtr;
<Use dp to access wave data.>
result = MDStoreDPDataInNumericWave(wavH, dPtr);         // Store data.
DisposePtr((Ptr)dPtr);
```

The data in the buffer is stored in row/column/layer/chunk order. This is explained in more detail on page 216 under **Organization of Numeric Data in Memory**.

Since the data in the buffer is always double-precision, regardless of the numeric type of the wave, this method relieves you of having to deal with multiple data types and will continue to work with future versions of Igor that support additional numeric types. It will also work with future versions of Igor that use a different organization of data in memory.

The disadvantage of this method is that it requires additional memory to hold the temporary copy of the wave data. You also pay a small speed penalty for copying the data.

If you are comfortable with memory allocation and deallocation and with pointers to double-precision data and if you are not willing to deal with the complexity of the direct access method (described below), then the temporary storage method is recommended for you.

When storing into an integer wave, MDStoreDPDataInNumericWave truncates the value that you are storing. If you want, you can do rounding before calling MDStoreDPDataInNumericWave .

### The Direct Access Method

The fastest and most difficult method is to use the MDAccessNumericWaveData routine to find the offset from the start of the wave handle to the numeric data. For example:

```
long dataOffset;
double* dp;
int hState;
int result;

if (result=MDAccessNumericWaveData(wavH, kMDWaveAccessMode0, &dataOffset))
   return result;
hState = MoveLockHandle(wavH);   // Lock handle so data won't move.
dp = (double*)((char*)(*wavH) + dataOffset);    // DEREFERENCE
```

At this point, dp points to the numeric data in the wave. The data in the buffer is stored in row/column/layer/chunk order. This is explained in more detail on page 216 under **Organization of Numeric Data in Memory**. See Chapter 13 for an example illustrating the use of MDAccessNumericData.

There are three difficulties in using the direct access method.

First, you need to lock the wave handle so that the data block to which it refers can not move in memory while you are pointing to it. This locking is not necessary if you are absolutely certain that you will do nothing to cause the blocks of memory in the heap to move. Unless you are doing something quite simple with the data, it is usually difficult to be certain that this is the case and will remain the case as you change the code. Thus, locking the handle, using MoveLockHandle, is recommended. It is critical to remember to restore the state of the handle, using HSetState, when you are finished with it so that it doesn't remain locked, clogging the heap.

The second and greatest difficulty is that you need to take into account the data type of the wave. Igor numeric waves can be one of the eight numeric types listed above. You need a different type of pointer to deal with each of these data types. This is illustrated in the example for MDAccessNumericWaveData in the WaveAccess sample XOP. If you are using C++, you can use a template to handle all of the data types.

The final difficulty is that, because you are pointing directly to the wave data, you need to know the layout of the data in memory. If a future version of Igor changes this layout (not very likely), your XOP will no longer work. Your XOP will not crash in that event, because the MDAccessNumericWaveData routine will see that you have passed it "kMDWaveAccess-Mode0". It will deduce from this that your XOP is treating the data in a way which, in the future version of Igor, is no longer appropriate. Thus, MDAccessNumericWaveData will return a non-zero error code result which your XOP will return to Igor. You will see an error alert in Igor

telling you that your XOP is obsolete and needs to be updated. Using the WaveData XOP-Support routine is functionally equivalent to using MDAccessNumericWaveData except that WaveData does not provide this compatibility check.

Because of the difficulties listed here, we recommend that you use the MDAccessNumeric-WaveData wave access method only in cases where you need the absolute top speed with minimal memory requirements. For many applications, it may be reasonable for you to support only some of the Igor numeric types, typically NT_FP64 and NT_FP32.

### Speed Comparisons

The WaveAccess sample XOP implements routines that fill a 3D wave using each of the wave access methods described above. To try them, compile the WaveAccess XOP and install it in the Igor Extensions folder. On Macintosh, make sure there's plenty of memory allocated to Igor Pro. Launch Igor Pro and execute the following commands:

```
Make/N=(50,50,50) wave3D
Variable timerRefNum

// Enter the next three lines as one command line
timerRefNum = StartMSTimer
WAFill3DWaveDirectMethod(wave3D)
Print StopMSTimer(timerRefNum)/1e6
```

Repeat replacing WAFill3DWaveDirectMethod with WAFill3DWavePointMethod and WAFill3DWaveStorageMethod.

The following times were recorded for filling a 50x50x50 double-precision wave.

| System | Direct Access | Temp Storage | Point Access |
|---|---|---|---|
| IMac 800MHz, Mac OS X | 0.0157 s | 0.0311 s | 0.0346 s |
| G4, 1250MHz, Mac OS 9.2 | 0.0142 | 0.0231 | 0.0211 |
| Generic PC, 800MHz, Windows 2000 | 0.0071 | 0.0211 | 0.0591 |

## Organization of Numeric Data in Memory

Using the direct or temporary storage wave access methods provides you with a pointer to wave data in memory. You need to know how the data is organized so that you can access it in a meaningful way.

The pointer that you get using the direct access method is a pointer to the actual wave data in the block of memory referenced by the wave handle. The pointer that you get using the temporary storage method is a pointer to a copy of the data. Data is organized in the same way in each of these methods.

Numeric wave data is stored contiguously in one of the supported data types. To access a particular element, you need to know the number of elements in each dimension. To find this, you must call MDGetWaveDimensions. This returns the number of used dimensions in the wave and an array of dimension lengths. The dimension lengths are interpreted as follows:

| | | |
|---|---|---|
| dimensionSizes[ROWS] | Number of rows in a column | |
| dimensionSizes[COLUMNS] | Number of columns in a layer | Zero for <2D waves |
| dimensionSizes[LAYERS] | Number of layers in a chunk | Zero for <3D waves |
| dimensionSizes[CHUNKS] | Number of chunks in the wave | Zero for <4D waves |

The symbols ROWS, COLUMNS, LAYERS and CHUNKS are defined in IgorXOP.h as 0, 1, 2 and 3 respectively.

For a wave of n dimensions, dimensionSizes[0..n-1] will be non-zero and dimensionSizes[n] will be zero.

The data is stored in row/column/layer/chunk order. This means that, as you step linearly through memory one point at a time, you first pass the value for each row in the first column. At the end of the first column, you reach the start of the second column. After you have passed the data for each column in the first layer, you reach the data for the first column in the second layer. After you have passed the data for each layer, you reach the data for the first layer of the second chunk.

If the data is complex, the real and imaginary part of each data point are stored contiguously, with the real part first.

## Accessing Text Wave Data

This section describes how Igor stores text data and how you can use the XOPSupport wave access routines to access the data. The routines mentioned here are described in detail in Chapter 13. The WaveAccess sample XOP illustrates how to use them.

Each element of a text wave can contain any number of characters. There are no illegal characters.

Igor stores all of the data for a text wave in the block of memory referenced by the wave handle, after the wave structure information. The wave structure contains another handle in which Igor stores indices. The indices tell Igor where the text for any given element in the text wave begins.

You should not attempt to access the text data or the text indices directly. Instead, use the two XOPSupport routines provide for getting and setting the contents of an element of a text wave. For example, here is a section of the WAModifyTextWave routine in WaveAccess.c:

```
indices[1] = column;
for(row=0; row<numRows; row++) {
   indices[0] = row;
   if (result = MDGetTextWavePointValue(wavH, indices, textH))
      goto done;
   <Modify the text in textH>;
   if (result = MDSetTextWavePointValue(wavH, indices, textH))
      goto done;
}
```

MDGetTextWavePointValue gets the contents of a particular element of the text wave which in this example is a 2D wave. The contents are returned via the pre-existing textH handle. textH is allocated by and belongs to the calling XOP, not to Igor.

textH contains just the characters in the specified element of the text wave. It does not contain a count byte, a trailing null character or any other information. To find the number of characters in the element, use GetHandleSize. If you want to treat the contents of the handle as a C string, you must null-terminate it first. Remember to remove the null terminator if you pass this handle back to Igor. You also must lock textH, using MoveLockHandle, while using the text if there is any chance of causing the blocks of memory in the heap to move. If you lock textH, you must remember to restore its locked/unlocked state, using HSetState.

If you can tolerate setting a maximum length for an element, you can use the XOPSupport GetCStringFromHandle and PutCStringInHandle to move the text between the handle and a C string which is easier to deal with. Using this technique, you don’t need to worry about adding or removing null terminators or locking or unlocking handles.

MDSetTextWavePointValue sets the specified element of the text wave. textH still belongs to the calling XOP. Igor just copies the contents of textH into the wave.

For dealing with the entire contents of large text waves, MDGetTextWavePointValue and MDSetTextWavePointValue may be too slow. When running with Igor Pro 5.04 or later you can use the faster but more complicated GetTextWaveData and SetTextWaveData routines which are described in Chapter 13.

For dealing with the entire contents of large text waves, MDGetTextWavePointValue and MDSetTextWavePointValue may be too slow. When running with Igor Pro 5.04 or later you can use the faster but more complicated GetTextWaveData and SetTextWaveData routines which are described in Chapter 13.

## Wave Scaling and Units

A wave has units and scaling for each dimension. In addition, it has units and full scale values for the data stored in the wave. Consider the following example of a 1D wave which stores voltage versus time:

| Row Index | X Index / X Units | Data / Data Units |
|---|---|---|
| 0 | 0.0 s | 3.57 v |
| 1 | 0.1 s | 3.21 v |
| 2 | 0.2 s | 2.97 v |
| 3 | 0.3 s | 2.73 v |
| 4 | 0.4 s | 2.44 v |
| 5 | 0.5 s | 2.03 v |

Data full scale = -10, 10

For a 1D wave, the row indices are also called point numbers. The X indices are calculated by Igor from the row indices using the scaling information for dimension 0 of the wave. You can set this scaling using a "SetScale x" command from within Igor and using the MDSetWaveScaling XOPSupport routine from an XOP. You can set the X units using a "SetScale x" command from within Igor and using the MDSetWaveUnits operation XOPSupport routine from an XOP.

You can set the data units and data full scale values using a "SetScale d" command from within Igor and using the SetWaveUnits and SetWaveScaling XOPSupport routines from an XOP. The data full scale is not analogous to the X scaling information. It merely is a record of the nominal full scale value for the data. For example, if you acquired the data using a digital oscilloscope set to the +/- 10V range, you can record this in the data full scale property of the wave. If this is not appropriate for your data, you can ignore the data full scale.

Considering a 2D wave, we see that the concepts of row indices, X indices and X units are extended to 2 dimensions, yielding column indices, Y indices and Y units.

| | | 0 | 1 | 2 | ← Column Index |
|---|---|---|---|---|---|
| | | 0.0 m | 2.5 m | 5.0 m | ← Y Index and Units |
| 0 | 0.0 s | 3.57 v | 8.24 v | 0.14 v | |
| 1 | 0.1 s | 3.21 v | 8.44 v | 0.26 v | |
| 2 | 0.2 s | 2.97 v | 8.76 v | 0.33 v | |
| 3 | 0.3 s | 2.73 v | 8.94 v | 0.47 v | |
| 4 | 0.4 s | 2.44 v | 9.31 v | 0.52 v | |
| 5 | 0.5 s | 2.03 v | 9.45 v | 0.55 v | |
| Row Index | X Index / X Units | Data / Data Units | Data / Data Units | Data / Data Units | |

Data full scale = -10, 10

You can set both the X (row) and Y (column) units and scaling using a SetScale command from within Igor and using the MDSetWaveUnits and MDSetWaveScaling XOPSupport routines from an XOP. If we extend to three dimensions, we add Z (layer) units and scaling and still use the same routines to set them.

Note that, as you go from 1D to 2D, you still have only one data units string and one data full scale. This is true no matter how many dimensions you use.

Prior to Igor Pro 3.0, when waves were restricted to one dimension, we spoke of a wave as having X units and Y units. When we extended Igor to support multi-dimensional waves, we realized that the term Y units was incorrect. This term should have been reserved for the units of the column dimension in multi-dimensional waves. For a 1D wave, we now speak of X (row) units and data units. For a 2D wave, we speak of X (row) units, Y (column) units and data units.

Because of this history, the WaveUnits and SetWaveUnits XOPSupport routines deal with the X units of the wave and the data units of the wave. The newer MDGetWaveUnits and MDSetWaveUnits routines deal with the units of all dimensions, including the X units and data units.

A similar situation holds for the WaveScaling and SetWaveScaling calls. These calls get or set the wave's X scaling and its data full scale values. The new MDGetWaveScaling and MDSetWaveScaling routines deal with the scaling of all dimensions, including the X dimension, and can also deal with the data full scale.

# Numeric and String Variables

This section decribes accessing variables other than an Operation Handler operation's runtime output variables (e.g., V_flag). For information on setting runtime output variables see **Runtime Output Variables** on page 174. Except for this technique, it is not possible for an XOP to set a function local variable. This section pertains to global variables and macro local variables.

Unlike waves, global and macro local variables are not defined by handles. Instead, Igor maintains an internal table of variable names and values.

A numeric variable is always double-precision real (NT_FP64) or double-precision complex (NT_FP64 | NT_CMPLX).

The content of a string variable is stored in a block of memory referenced by a handle but this is not the same as a wave handle. In the case of the wave handle, the handle contains the wave structure as well as the wave data. Consequently, the wave handle completely defines a wave. In the case of a string variable, the handle contains just the variable's data. It does not contain the string variable's name or a reference to its data folder.

A string handle contains the string's text, with neither a count byte nor a trailing null byte. Use GetHandleSize to find the number of characters in the string. To use C string functions on this text you need to copy it to a local buffer and null-terminate it or add a null terminator to the handle and lock the handle. If you pass the handle back to Igor, you must remove the null terminator and unlock the handle.

If you can tolerate setting a maximum length for a string, you can use the XOPSupport GetCStringFromHandle and PutCStringInHandle to move the text between the handle and a C string which is easier to deal with. Using this technique, you don't need to worry about adding or removing null terminators or locking or unlocking handles.

The XOPSupport library provides routines to create variables, kill variables, get the value of variables and set the value of variables. All of these routines require that you pass the variable name to Igor. Some routines also allow you to specify the data folder in which the variable is to be created or found. Those routines that do not allow you to specify the data folder work in the current data folder.

## Routines for Accessing Variables

The XOPSupport routines for dealing with variables are described in detail in Chapter 13. Here is a summary of the commonly-used routines.

### Creating Variables

| Routine | Description |
|---|---|
| Variable | Creates an Igor numeric or string variable. |
| SetIgorIntVar | Creates if necessary and then sets a numeric variable to an integer value. |
| SetIgorFloatingVar | Creates if necessary and then sets a numeric variable to a floating point value. |
| SetIgorComplexVar | Creates if necessary and then sets a numeric variable to a complex floating point value. |
| SetIgorStringVar | Creates if necessary and then sets a string variable. |

### Getting Variable Contents

| Routine | Description |
|---|---|
| FetchNumVar | Gets the value of a numeric variable. |
| FetchStrVar | Gets up to 255 characters from a string variable. |
| FetchStrHandle | Gets any number of characters from a string variable. |
| GetDataFolderObject | Gets the value of numeric or string variable. |

### Setting Variable Contents

| Routine | Description |
|---|---|
| StoreNumVar | Sets the value of a numeric variable. |
| SetIgorIntVar | Creates if necessary and then sets a numeric variable to an integer value. |
| FetchStrHandle | Gets any number of characters from a string variable. |
| SetIgorFloatingVar | Creates if necessary and then sets a numeric variable to a floating point value. |
| SetIgorComplexVar | Creates if necessary and then sets a numeric variable to a complex floating point value. |
| SetIgorStringVar | Creates if necessary and then sets a string variable. |
| SetFileLoaderOutputVariables | Specialized for file-loader XOPs. |
| SetOperationFileLoaderOutputVariables | Specialized for file-loader XOPs. Called from direct external operations only. |
| SetDataFolderObject | Sets the value of numeric or string variable. |

## Example

Here is a simple example that illustrates creating numeric and string variables and getting and setting their values. The underlined items are functions, constants or structures defined in the XOPSupport library and headers.

```
static int
MakeVariables(void)
{
   char varName[MAX_OBJ_NAME+1];
   double realVal, imagVal;
   char strValue[256];
   int err;

   strcpy(varName, "numVar0");
   realVal = 3.14159;
   if (err= SetIgorFloatingVar(varName, &realVal, 1)) // Create and set
      return err;
   if (FetchNumVar(varName, &realVal, &imagVal)==-1)  // Fetch value
      return EXPECTED_VARNAME;

   strcpy(varName, "strVar0");
   strcpy(strValue, "This is a string variable.");
   if (err= SetIgorStringVar(varName, strValue, 1))   // Create and set
      return err;
   if (err= FetchStrVar(varName, strValue))           // Fetch value
      return err;

   return 0;
}
```

The last parameter in the SetIgorFloatingVar and SetIgorStringVar routines requests that Igor make the variables global. If this parameter were zero and if our routine were called during the execution of a macro, Igor would make the variables local to the macro. You can not create a local variable in a user-defined function at runtime as these are created by Igor when the function is compiled.

For a discussion of creating output variables from an external operation, analogous to the V_ variables created by many Igor operations, see **Runtime Output Variables** on page 174.

The string part of this example uses routines that take and return C strings. This is easier than dealing with the actual string variable handle and is fine as long as the string variable contains 255 or fewer characters.

This example works in the current data folder. In most cases, this is the appropriate behavior.

# Dealing With Data Folders

Data folders provide a way to store waves, numeric variables and string variables in a hierarchy. Data folders are analogous to file system folders. However, data folders are maintained entirely by Igor.

If you save an Igor experiment in a packed experiment  file, the entire hierarchy is saved in the packed file. If you save an experiment in an unpacked file, Igor creates file system folders to mirror the data folder hierarchy. Because these file system folders are created only when you save an experiment unpacked, you should not assume that a file system folder exists to mirror any given Igor data folder. You should think of data folders as a hierarchy maintained by Igor in memory and saved to disk for storage purposes.

Igor stores information about each data folders in a DataFolderHandle. Each XOPSupport routine that deals with data folders returns a DataFolderHandle to you and/or requires a DataFolder-Handle from you. You have no way of knowing or need to know the details of DataFolder-Handles. They are black boxes. You merely receive them from Igor and pass them back to Igor.

When you get a DataFolderHandle from Igor, you should use it and forget about it. The handle belongs to Igor so you should not modify or dispose it. Also, you should not store it long-term because it will become invalid if the user kills the folder or opens a new experiment.

## Routines for Accessing Data Folders

The XOPSupport routines for dealing with data folders are described in detail in Chapter 13. Most of them exist for the benefit of the Igor Data Browser, which is an XOP, and for other sophisticated XOPs. Here are the data folder routines that are most likely to be of use for a typical data-folder-aware XOP.

### Getting a Handle to a Data Folder

| Routine | Description |
|---|---|
| GetDataFolderAndName | Returns a data folder handle and an object name. |
| GetDataFolder | Returns a data folder handle. |
| GetRootDataFolder | Returns a handle to the root data folder. |
| GetCurrentDataFolder | Returns a handle to the current data folder. |
| GetNamedDataFolder | Returns a handle to a data folder, given its name or path. |
| GetWavesDataFolder | Returns a handle to the data folder containing a particular wave. |
| GetParentDataFolder | Returns a handle to the parent of the specified data folder. |
| GetNumChildDataFolders | Returns the number of child data folders within a particular data folder. |
| GetIndexedChildDataFolder | Returns a handle to the specified data folder within a particular data folder. |

### Creating and Killing a Data Folder

| Routine | Description |
|---|---|
| NewDataFolder | Creates a new data folder. |
| DuplicateDataFolder | Duplicates an existing data folder and its contents. |
| KillDataFolder | Kills a data folder. |

### Accessing Objects in a Data Folder

| Routine | Description |
|---|---|
| GetNumDataFoldersObjects | Returns the number of objects (waves, numeric variables, string variables, data folders) in a particular data folder. |
| GetIndexedDataFolderObject | Returns the name of a particular object. |
| GetDataFolderObject | Returns information about a particular object. |
| SetDataFolderObject | Sets information about a particular object. |
| KillDataFolderObject | Kills a particular object. |
| DuplicateDataFolderObject | Duplicates a particular object. |

If you ignore the data-folder-related XOPSupport routines, your XOP will work in the current data folder. This is the appropriate behavior for many applications. Also, no special effort is required to work with existing waves in any data folder. Once you have the wave handle, you can manipulate the wave without knowing which data folder it resides in.

## Data Folder Conventions

One of the points of using data folders is to reduce clutter. If your XOP requires a lot of private waves and/or variables, you should create a data folder to contain your private storage. By convention, data folders for this purpose are stored in another data folder named Packages which is in the root. This is discussed in the programming section of the Igor Pro manual.

Here is some code that will create a data folder for your private storage inside the Packages data folder.

```
static int
GetPrivateDataFolderHandle(char* dataFolderName, DataFolderHandle* dfHPtr)
{
   DataFolderHandle rootDFH, packagesDFH;
   int err;

   if (err = GetRootDataFolder(0, &rootDFH))
      return err;

   if (err = GetNamedDataFolder(rootDFH, "Packages", &packagesDFH)) {
      // We need to create Packages.
      if (err = NewDataFolder(rootDFH, "Packages", &packagesDFH))
         return err;
   }

   if (err = GetNamedDataFolder(packagesDFH, dataFolderName, dfHPtr)) {
      // We need to create our private data folder.
      if (err = NewDataFolder(packagesDFH, dataFolderName, dfHPtr))
         return err;
   }

   return 0;
}
```

You would call this routine like this:

```
DataFolderHandle privateStorageDFH;
if (err = GetPrivateDataFolderHandle("MyPrivateData", &privateStorageDFH))
   return err;
```

Of course, you should choose a name for your data folder that will make it clear what it is for and is specific enough to avoid conflicts with other packages.

Because the user can kill a data folder, intentionally or inadvertently, you should not store the data folder handle but instead should obtain it from Igor, as shown above, each time you need to use it. Another option is to use the GetDataFolderIDNumber and GetDataFolderByIDNumber routines, listed in Chapter 13.

Chapter 8

# Adding Menus and Menu Items


Overview ... 231
Menu Manager Routines ... 232
MenuHandles ... 232
Summary of Menu Manager Routines ... 233
Adding Menus and Menu Items ... 234
Adding a Main Menu ... 235
Adding Menu Items to Igor Menus ... 236
Adding Submenus ... 237
Menu IDs Versus Resource IDs ... 238
Responding to Menu Selections ... 239
Determining Which Menu Was Chosen ... 239
Getting Your Menu Handle ... 240
Determining Which Menu Item Was Chosen ... 241
Detecting the First Time Your Menu Item is Chosen ... 241
Enabling and Disabling Menu Items ... 242
Enabling and Disabling XOP Menu Items ... 242
Enabling and Disabling Igor Menu Items ... 243
Advanced XOP Menu Issues ... 245
Avoiding Menu ID Conflicts ... 245
Dialog Popup Menus ... 245
Menu Bars in Windows ... 246

# Overview

When Igor starts up, it examines each XOP file looking for resources that define menus, menu items and submenus that the XOP adds to Igor.

An XOP may elect to add no menus or menu items to Igor. For example, an XOP that does no more than add an operation or function to Igor does not need a menu or menu item. The XOP should have none of the resources discussed below in its resource fork.

An XOP may need just one simple menu item in an existing Igor menu. For example, an XOP that adds a number-crunching operation to Igor might want to simply add one menu item to Igor's Analysis menu.

An XOP may need more than one simple menu item in an existing Igor menu. For example, an XOP that allows Igor to load and save a particular file format might want to add one menu item to Igor's Load Waves menu and another menu item to Igor's Save Waves menu.

An XOP may want to add a menu item to a built-in Igor menu but might want that menu item to have a submenu.

An XOP may want to have its own menu in the main menu bar and might want that menu to have submenus. This would be appropriate for an elaborate XOP that adds major functionality to Igor.

The MenuXOP1 sample XOP illustrates all of the above. It also shows how an XOP can determine which of its menu items has been selected and how to access each menu item to enable or disable it.

This chapter discusses many different ways for an XOP to use menus, submenus, and menu items. If your XOP merely adds a single unchanging menu item to Igor, you can skip down to the section **Adding Menu Items to Igor Menus**. This explains how to use an XMI1 resource to add a menu item. Then add code to your XOPEntry routine to respond when Igor sends a MENUITEM message. That is all you need to know to use a single menu item.

# Menu Manager Routines

This section contains background information that most XOP programmers will not need to know. You need to know this only if your XOP adds, removes, or modifies menus or menu items.

Igor uses Macintosh Menu Manager routines to implement its menus. This is true even on Windows, where Igor emulates the Macintosh Menu Manager. Because XOP menu items are integrated with Igor menu items, XOPs that add menu items must also use Macintosh Menu Manager routines, even when running on Windows.

On Macintosh, the menu manager routines previously were provided by the Mac OS. In the Carbon API, Apple changed the names of all of the routines. In order to support old XOPs, these routines are now supported through the XOPSupport library. On Windows, they are provided by the IGOR.lib library, with which all XOPs link.

This section does not describe the Macintosh Menu Manager completely. Instead, it describes a very simplified version of the menu manager and how an XOP typically uses menu manager routines to manipulate its menus. The routines described below can be used on both Macintosh and Windows.

## MenuHandles

For each menu, there is a menu handle, of type MenuHandle, that contains all of the information for the menu. For XOPs that add one or more menus to Igor, the menus and menu handles are created automatically by Igor at launch time, when Igor examines the XOP's XMN1 and XSM1 resources. These resources are described on page 234 in the section **Adding Menus and Menu Items**.

You need not be concerned with the exact contents of a menu handle. To operate on it, you will pass it to the menu manager routines. For example, to disable a menu item, you pass the menu handle to the DisableItem routine. To set the text of a menu item, you pass the menu handle to the setmenuitemtext routine. Here is an example.

```
MenuHandle mH;
mH = ResourceMenuIDToMenuHandle(100);         // Get our menu handle.
if (mH != NULL) {                             // Always test before using.
   DisableItem(mH, 1);                        // Disable the first item.
   setmenuitemtext(mH, 2, "Capture Image");   // Set text of second item.
}
```

The ResourceMenuIDToMenuHandle routine is not a menu manager routine. It is a regular XOPSupport routine and is described on page 240 under **Getting Your Menu Handle**.

## Summary of Menu Manager Routines

Here are the menu manager routines that can be used for Macintosh and Windows XOPs. See **Emulated Menu Management Routines** on page 502 for a detailed description of these routines.

| Routine | What It Does |
|---|---|
| CountMItems | Returns the number of menu items in a menu. |
| DeleteMenuItem | Deletes a menu item from a menu. |
| insertmenuitem | Inserts a new item into a menu. |
| appendmenu | Adds a new item to the end of a menu. |
| getmenuitemtext | Retrieves the text for a menu item. |
| setmenuitemtext | Sets the text for a menu item. |
| DisableItem | Disables (grays out) a menu item. |
| EnableItem | Enables a menu item. |
| CheckItem | Adds or removes a check mark from a menu item. |

# Adding Menus and Menu Items

An XOP can add a main menu bar menu, any number of menu items to built-in Igor menus, and submenus to any menu items. Most XOPs will use it just to add a single menu item, if at all.

This method involves the use of the following resources.

| Resource | What It Is Used For |
|---|---|
| XMN1 1100 | Adds a menu to the main menu bar. |
| XSM1 1100 | Adds submenus to XOP menu items. |
| XMI1 1100 | Adds menu items to Igor menus. |

The MenuXOP1 sample XOP illustrates the use of these resources which are described in detail below. See the MenuXOP1.r and MenuXOP1WinCustom.rc files in particular.

There are three limitations that you must be aware of.

First, the main menu bar can hold only a finite number of menus. It is possible for menus to be inaccessible because XOPs have added too many of them. You need to decide if your XOP should take space on the main menu bar or if it should merely add a submenu to a built-in Igor menu.

Second, there are 100 menu IDs reserved for all XOP main menus and 50 menu IDs reserved for all XOP submenus. It is unlikely that either of these limits will be reached but, since there are a limited number, you should use no more main menus or submenus than necessary in your XOP.

Finally, you must be careful when you create the resources in your XOP that describe your XOP's menus to Igor. It's not difficult to cause a crash by specifying the wrong menu IDs.

The following sections explain how Igor determines what menus, submenus and menu items your XOP adds.

## Adding a Main Menu

Igor looks for an XMN1 1100 resource. XMN means “XOP menu”. This resource specifies the menu or menus, if any that the XOP adds to the main menu bar.

Here is an example of an XMN1 resource:

```
// Macintosh
resource 'XMN1' (1100) {
   {
      100,            // menuResID: Menu resource ID is 100.
      1,              // menuFlags: Show menu when Igor is launched.
   }
};

// Windows
1100 XMN1
BEGIN
   IDR_MENU1,         // menuResID: Menu resource ID is IDR_MENU1.
   1,                 // menuFlags: Show menu when Igor is launched.
   0,                 // 0 required to terminate the resource.
END
```

The first field, called menuResID, is the resource ID of the MENU resource in the XOP’s resource fork for the menu to be added to Igor’s main menu bar. The symbol IDR_MENU1 in the Windows resource is defined in the file resource.h, which is created by the Visual C++ resource editor.

The second field, called menuFlags, consists of bitwise flags defined in XOP.h as follows:

```
#define SHOW_MENU_AT_LAUNCH 1    // bit 0
#define SHOW_MENU_WHEN_ACTIVE 2  // bit 1
```

If bit 0 of the menuFlags field is set then the menu is appended to the main menu bar when Igor is launched. This is appropriate for a menu that is to be a permanent fixture on the menu bar. If bit 0 is cleared then the menu is *not* appended at this time.

If bit 1 of the menuFlags field is set then Igor automatically inserts the menu in the main menu bar when your XOP’s window is active and removes it when your XOP’s window is deactivated.

If neither bit is set then your XOP can show and hide the menu on its own. This is illustrated by the MenuXOP1 sample XOP.

The remaining bits are reserved and must be set to zero.

The resource shown adds one menu to Igor's main menu bar. To add additional menus you would add additional pairs of fields to the resource.

## Adding Menu Items to Igor Menus

Igor looks for a resource of type XMI1 1100 that specifies menu items to be added to built-in Igor menus. XMI means "XOP menu item".

Here is an example of an XMI1 resource:

```
resource 'XMI1' (1100) {    // Macintosh, in MenuXOP1.r.
   {
      8,                    // menuID: Add item to menu with ID=8.
      "MenuXOP1 Misc1",     // itemText: This is text for added menu item.
      0,                    // subMenuResID: This item has no submenu.
      0,                    // itemFlags: Flags field.
   }
};

1100 XMI1                   // Windows, in MenuXOP1WinCustom.rc.
BEGIN
   8,                       // menuID: Add item to menu with ID=8.
   "MenuXOP1 Misc1\0",      // itemText: This is text for added menu item.
   0,                       // subMenuResID: This item has no submenu.
   0,                       // itemFlags: Flags field.
   0,                       // 0 required to terminate the resource.
END
```

The first field, called menuID, is the menu ID of the built-in Igor menu to which the menu item should be attached. The file IgorXOP.h gives the menu ID's of all Igor menus.

The second field, called itemText, is a string containing the text for the added menu item.

The third field, called subMenuResID, is the resource ID of the MENU resource in the XOP's resource fork for the submenu to be attached to the menu item or 0 for no submenu.

The fourth field, called itemFlags, consists of bitwise flags defined in XOP.h which tell Igor under what conditions the menu item should be enabled or disabled. See **Enabling and Disabling XOP Menu Items** on page 242.

The resource shown adds one menu item to a built-in Igor menu. To add additional menu items you would add additional sets of fields to the resource. See MenuXOP1.r and MenuXOP1WinCustom.rc for examples.

## Adding Submenus

Igor looks for a resource of type XSM1 1100. XSM means “XOP submenu”. This resource specifies the submenu or submenus that the XOP wants to attach to menu items in menus that belong to the XOP (not built-in Igor menus).

Here is an example of an XSM1 resource:

```
// Macintosh
resource 'XSM1' (1100) {
   {
      101,        // subMenuResID: Add submenu with resource ID 101
      100,        // mainMenuResID: to menu with resource ID 100
      1,          // mainMenuItemNum: to item 1 of main menu
   }
};

// Windows
1100 XSM1
BEGIN
   IDR_MENU2,     // subMenuResID: Add submenu with resource ID IDR_MENU2
   IDR_MENU1,     // mainMenuResID: to menu with resource ID IDR_MENU1
   1,             // mainMenuItemNum: to item 1 of menu.
   0,             // 0 required to terminate the resource.
END
```

The first field, called subMenuResID, is the resource ID of the MENU resource in the XOP’s resource fork for the submenu to be attached to a menu item.

The second field, called mainMenuResID, is the resource ID of the MENU resource in the XOP’s resource fork for the menu to which the submenu is to be attached.

The third field, called mainMenuItemNum, is the item number in that menu to which the submenu is to be attached.

The symbols IDR_MENU1 and IDR_MENU2 in the Windows resource are defined in the file resource.h, which is created by the Visual C++ resource editor.

The resource shown adds one submenu to an XOP menu. To add additional submenus you would add additional sets of fields to the resource.

mainMenuResID and mainMenuItemNum normally will refer to a menu declared in the XMN1 resource but can also refer to a menu declared in a previous record of the XSM1 resource. This allows submenus to have submenus.

## Menu IDs Versus Resource IDs

On Macintosh only, when adding a MENU resource to your XOP, make sure that the menu ID and the resource ID are the same. The resource ID is used by the Macintosh Resource Manager but the Macintosh Menu Manager cares about the menu ID, not the resource ID. Here is a correct Rez resource description:

```
resource 'MENU' (100) {        // Resource ID is 100
   100,                        // Menu ID is same as resource ID
   textMenuProc,
   0xffffffff,
   enabled,
   "VDT",
   {
      . . .
   }
};
```

# Responding to Menu Selections

When the user chooses an XOP's menu item, Igor loads the XOP into memory (if it's not already loaded) and sends a MENUITEM message to the XOP, passing it a menu ID and an item number that specify which item in which menu was chosen. The XOP responds by performing the appropriate action. It returns an error code to Igor, using the SetXOPResult XOPSupport routine, which indicates any problems encountered. If an error did occur, Igor presents a dialog with an appropriate error message.

Igor sends the MENUITEM message to Windows XOPs as well as to Macintosh XOPs. Windows XOPs will not receive a WM_COMMAND message from the Windows OS when the XOP's menu item is chosen.

How you interpret the menuID and itemNumber parameters that come with the MENUITEM message depends on how your XOP is set up.

## Determining Which Menu Was Chosen

If your XOP adds just one menu item to Igor then matters are simple. When you receive the MENUITEM message, you know what the user chose without considering the menuID and itemNumber parameters.

Matters become more complex if your XOP adds more than one menu item. To understand this you need to know a bit about menu IDs. A menu ID is a number associated with a particular menu in a program. When the user selects a menu item, the Macintosh Menu Manager tells the program which menu was selected by passing it a menu ID. Because the Windows version of Igor uses Macintosh emulation for menus, even Windows XOP menus have menu IDs.

In a standalone application, a menu ID for a given menu is normally the same as the resource ID for the resource from which the menu came. For example, Igor's Misc menu has a menu ID of 8 and is defined by a MENU resource with resource ID=8 in Igor's resource fork.

For XOPs, matters are slightly complicated. There could be several XOPs with MENU resources with ID=100 in their resource forks. Obviously, when all of these menus are loaded into memory, they can not all have menu IDs of 100. To get around this, Igor assigns new menu IDs to each XOP menu. This occurs when Igor inspects each XOP's resources at the time Igor is launched.

The original resource ID of the XOP's menu is called the "resource menu ID". The ID assigned by Igor is called the "actual menu ID".

When you write your XOP, you do not know the actual menu ID that your menu or menus will have when your XOP is running. Therefore, you need a way to translate between the actual menu ID and the resource menu ID. The XOP Toolkit provides two routines to handle this:

```
int
ResourceToActualMenuID(resourceMenuID)
int resourceMenuID;
```

Given the ID of a MENU resource in the XOP's resource fork, returns the actual menu ID of that resource in memory.

Returns 0 if XOP did not add this menu to Igor menu.

```
int
ActualToResourceMenuID(menuID)
int menuID;
```

Given the actual ID of a menu in memory, returns the resource ID of the MENU resource in the XOP's resource fork.

Returns 0 if XOP did not add this menu to Igor menu.

Imagine that your XOP adds one menu to Igor. When you get the MENUITEM message from Igor, you know that the menuID passed with that message is the actual menu ID for the XOP's menu so you don't need to use the ActualToResourceMenuID routine.

On the other hand, if your XOP adds two menus to Igor then, when you get the MENUITEM message, you do need to use the ActualToResourceMenuID routine to determine which of your two menus was selected.

In order to change your menu, for example, to disable one of its items, you need its menu handle. You can get the menu handle by calling GetMenuHandle or by calling ResourceMenuIDTo-MenuHandle. These are described in the next section. If you use GetMenuHandle, you need to use the ResourceToActualMenuID routine in order to get your menu's actual menu ID.

## Getting Your Menu Handle

To actually do anything with your menu, such as enable or disable an item, you need to get a menu handle. Once you have the actual menuID, you can get the menu handle using the Macintosh Menu Manager GetMenuHandle routine. *However*, if your menu is hidden, GetMenuHandle will return NULL. You should always be prepared to handle a NULL return value from GetMenuHandle.

The XOPSupport routine ResourceMenuIDToMenuHandle provides a way to get your menu handle even if it is hidden. Unlike GetMenuHandle, it takes the menu's resource ID, not its actual ID, as a parameter.

```
MenuHandle
ResourceMenuIDToMenuHandle(resourceMenuID)
int resourceMenuID;
```

Use this to get a handle to a menu or submenu that you have added to Igor. Do not use it to get a handle to built-in Igor menus. Unlike the Macintosh Toolbox GetMenuHandle routine, ResourceMenuIDToMenuHandle will return your menu handle even if it is hidden.

resourceMenuID is the resource ID of your menu in your resource fork.

## Determining Which Menu Item Was Chosen

When your XOP adds an item to a built-in Igor menu, you always know the menu ID for the menu since it is defined in IgorXOP.h and is fixed for all time. However, since the menu item is appended to the end of the menu, you will not know the item's item number when you write the XOP. The ResourceToActualItem and ActualToResourceItem routines translate between resource item numbers and actual item numbers. Menu item numbers start from one.

```
int
ResourceToActualItem(igorMenuID, resourceItemNumber)
int igorMenuID;
int resourceItemNumber;
```

Given the ID of a built-in Igor menu and the one-based number of a menu item specification in the XMI1 resource in the XOP's resource fork, returns the actual item number of that item in the Igor menu.

Returns 0 if the XOP did not add this menu item to Igor menu.

```
int
ActualToResourceItem(igorMenuID, actualItemNumber)
int igorMenuID;
int actualItemNumber;
```

Given the ID of a built-in Igor menu and the actual number of a menu item in the Igor menu, returns the one-based number of the specification in the XMI1 resource in the XOP's resource fork for that item.

Returns 0 if the XOP did not add this menu item to Igor menu.

## Detecting the First Time Your Menu Item is Chosen

You may want to know if your XOP was just loaded into memory and INITed when you receive the MENUITEM message so that you can do something special the first time. Use the GetXOPStatus XOPSupport routine and test the XOPINITING bit in the resulting status. If it is set then the user selecting your menu item caused your XOP to be loaded into memory. If it is not set then your XOP was already in memory and INITed when the user selected your item.

# Enabling and Disabling Menu Items

When the user clicks in the menu bar or presses a keyboard equivalent, Igor allows all XOPs that are in memory to enable or disable menu items.

If an XOP window is the active window, Igor Pro disables all built-in menu items that pertain to the active window (e.g., Cut, Copy, Paste). It enables all menu items that can be invoked no matter what the active window is (e.g., Miscellaneous Settings, Kill Waves, Curve Fitting). Then it sends the MENUENABLE message to each loaded XOP that adds a menu or menu item to Igor. As of Igor Pro 5, Igor sends the MENUENABLE message to the XOP that created the active window even if that XOP added no menu items. Each XOP can set its own menu items as it wishes. If its window is the active window, the XOP can also re-enable built-in Igor menu items that apply to the active window.

Igor sends the MENUENABLE message to Windows XOPs as well as to Macintosh XOPs. Windows XOPs will not receive a WM_INITMENU message from the Windows OS.

An XOP whose window is active can respond when the user selects a menu item. The menu item may be a built-in Igor menu item or it may be a custom item, created by the XOP. If the user selects a built-in Igor menu item and the XOP's window is the active window then Igor sends the XOP a message. Examples are the CUT, COPY, and PASTE messages. If the user selects an XOP's custom menu item then Igor sends the MENUITEM message.

Since the XOP can respond to built-in and XOP menu items, it needs a way to enable and disable these items.

## Enabling and Disabling XOP Menu Items

When you receive the MENUENABLE message you can enable or disable items added to the menu by your XOP. To do this, you need to determine the menu ID and item number of your menu item and then call the EnableItem or DisableItem menu manager routines. The section **Responding to Menu Selections** explains how to determine the menu ID and item number of your menu item.

There is another method which allows Igor to automatically enable and disable XOP menu items added to built-in Igor menus. This method involves the itemFlags field of the XMI1 1100 resource. The XOPTypes.r file defines the following bits for this field:

```
ITEM_REQUIRES_WAVES               ITEM_REQUIRES_GRAPH
ITEM_REQUIRES_TABLE               ITEM_REQUIRES_LAYOUT
ITEM_REQUIRES_GRAPH_OR_TABLE      ITEM_REQUIRES_TARGET
ITEM_REQUIRES_PANEL               ITEM_REQUIRES_NOTEBOOK
ITEM_REQUIRES_GRAPH_OR_PANEL      ITEM_REQUIRES_DRAW_WIN
ITEM_REQUIRES_PROC_WIN
```

For example, if you set the ITEM_REQUIRES_WAVES bit in the itemFlags field of your XMI1 1100 resource, Igor will automatically enable your item if one or more waves exists in the current data folder and automatically disable it if no waves exist in the current data folder.

For another example, if the ITEM_REQUIRES_GRAPH bit is set, Igor will enable your menu item only if the target window is a graph.

## Enabling and Disabling Igor Menu Items

If your XOP adds a window to Igor, you might want to enable and disable Igor menu items such as Cut and Copy when your window is active. This section describes how to do this.

The SetIgorMenuItem XOPSupport routine allows the XOP to enable or disable a built-in Igor menu item without referring to specific menu IDs or item numbers. Here is a description:

```
int
SetIgorMenuItem(message, enable, text, param)
int message;          // an Igor message code
int enable;           // 1 to enable the menu item, 0 to disable it
char* text;           // pointer to a C string or NULL
long param;           // normally not used and should be 0
```

For example, if the user selects the Copy menu item, Igor sends the XOP the COPY message. If the XOP wants to enable the Copy menu item, it would call Igor as follows:

```
SetIgorMenuItem(COPY, 1, NULL, 0);
```

The text parameter will normally be NULL. However, there are certain built-in Igor menus whose text can change. An example of this is the Undo item. An XOP which owns the active window can set the Undo item as follows:

```
SetIgorMenuItem(UNDO, 1, "Undo XOP-Specific Action", 0);
```

Igor will ignore the text parameter for menu items whose text is fixed, for example Copy. For menu items whose text is variable, if the text parameter is not NULL, then Igor will set the text of the menu item to the specified text.

There is currently only one case in which the param parameter is used. If the message is FIND, Igor needs to know if you want Find, Find Same or Find Selected Text. It looks at the param parameter for this which should be 1, 2 or 3, respectively. In all other cases, you must pass 0 for the param parameter.

SetIgorMenuItem returns 1 if there is a menu item corresponding to message or 0 if not. Normally you will have no need for this return value.

Here is some code showing how to enable Igor menu items.

```
static void
XOPMenuEnable(void)
{
	.
	.
	SetIgorMenuItem(COPY, 1, NULL, 0);
	SetIgorMenuItem(CUT, 1, NULL, 0);
	.
	.
}
```

# Advanced XOP Menu Issues

XOP menus and submenus created using the techniques described above in this chapter are permanent. Igor creates them at launch time and they persist until Igor quits.

Some very advanced XOPs may need to create menus on their own, for example to implement popup menus in a window or in a dialog. This is something that is commonly done on Macintosh but not on Windows, because Windows programs generally use combo boxes where Macintosh programs use popup menus. This section discusses issues that you must take into account if you create your own menus.

## Avoiding Menu ID Conflicts

When Igor creates XOP menus, it makes sure that each menu uses a unique menu ID. When an XOP creates its own menus, it needs to do the same thing. To avoid menu ID conflicts, XOPs should not create permanent menus. Instead, they should create menus, use them, and then delete them. For example, a popup menu in a window should be created when the user clicks and deleted when the use releases the mouse. Popup menus in dialogs should be created when the dialog is created and deleted when the dialog is closed. The dialog popup menu XOPSupport routines behave in this way so, if you use them to implement your dialog popup menus, you will avoid conflicts.

You can create a temporary menu by calling newmenu, which creates the menu at runtime or GetMenu, which load the menu from a resource. If you create a menu via newmenu, you must dispose it by calling DisposeMenu. You must call DisposeMenu once for each newmenu call. If you create a menu via GetMenu, you must dispose it by calling ReleaseMenu. You must call ReleaseMenu once for each GetMenu call.

In addition to deleting menus when you are finished with them, you must also restrict your menu IDs to ranges that Igor reserves for your use. The ranges 200-219 and 1100-1199 are reserved for XOP temporary menus. Because of Macintosh Menu Manager restrictions, submenus must fall in the range 200-219.

## Dialog Popup Menus

The recommended method for implementing a popup menu in a dialog is to call CreatePopMenu. CreatePopMenu uses the first free menu ID in the range 1150-1199 to create a temporary menu. The menu is disposed when you call KillPopMenus, at the end of the dialog.

For more information of dialog popup menus, see **Cross-Platform Dialog Popup Menus** on page 275

## Menu Bars in Windows

It would be nice if a Windows XOP could add a menu bar to its own window. However, this is not possible. Igor is a Multiple Document Interface (MDI) application. Windows does not allow you to add a menu bar to an MDI child window. It does allow you to add a menu bar to an "overlapped" window. However, overlapped windows do not behave correctly in MDI applications.

Chapter 9

# Adding Windows

Overview ........ 249
Adding a Simple Window on Macintosh ........ 250
Adding a Simple Window on Windows ........ 251
    CreateXOPWindowClass ........ 252
    CreateXOPWindow ........ 252
    DestroyXOPWindow ........ 252
    XOPWindowProc ........ 252
    Menus in Windows XOP Windows ........ 253
TU ("Text Utility") Windows ........ 254
Recursion Problems ........ 254
Igor Window Coordinates ........ 255
Adding XOP Target Windows ........ 257

# Overview

Your XOP can add one or more windows to Igor. Of course, doing this is much more involved than merely adding an operation or a function. If you are writing a Macintosh XOP, you will need to be familiar with the Macintosh Carbon API. If you are writing a Windows XOP, you will need to be familiar with the Win32 API. Programming with windows can raise many complex issues and debugging a program that creates a window can be difficult. You should not try to add a window to Igor unless you are an experienced programmer or you are willing to spend significant time learning.

An XOP can add a simple one-of-a-kind window. The simplest example of this is the WindowXOP1 XOP, which adds a window that merely displays a message and responds to menu selections. Another simple example is TUDemo, which uses the XOP Toolkit "text utility" routines (TU) to implement a simple text editing window. On the other end of the complexity scale is an XOP that adds a type of window to Igor, of which there may be an indefinite number of instances. An example of this is the Igor Pro Surface Plotter (the source code for which is not part of the XOP Toolkit).

Adding a window on Macintosh is quite different from adding a window on Windows. This stems from the very different ways in which these operating systems send messages to a window. The main difference is that a Macintosh XOP receives window-related messages (e.g., click, type, activate, update) from Igor while a Windows XOP receives these messages directly from the Windows OS.

The WindowXOP1 XOP illustrates the technique of implementing an XOP in two parts: a platform-independent part and a platform-dependent part. For the platform-dependent part, there are parallel source files for Macintosh and Windows. For example, the functions CreateXOPWindow and DestroyXOPWindow are defined in both WindowXOP1Mac.c and WindowXOP1Win.c, thus allowing WindowXOP1.c to call them on either platform.

Cross-platform XOPSupport routines that deal with windows take parameters of type XOP_WINDOW_REF. An XOP_WINDOW_REF is a WindowPtr on Macintosh and an HWND on Windows.

# Adding a Simple Window on Macintosh

To create a simple window on Macintosh, use the GetXOPWindow XOPSupport routine. This routine works much like the Macintosh GetNewWindow toolbox routine. However, GetXOP-Window sets the windowKind field of the resulting window record so that Igor can tell that the window belongs to your XOP. You must not change this field. In WindowXOP1, GetXOP-Window is called from CreateXOPWindow.

When you are finished with your window, use the Macintosh DisposeWindow routine to dispose it. In WindowXOP1, DisposeWindow is called from DestroyXOPWindow.

If your Macintosh XOP adds a window to Igor it will receive all of the window-oriented messages from Igor. You must respond properly to the UPDATE message (BeginUpdate . . . draw . . . EndUpdate) so that the window update event will not be generated over and over again. You should also set the cursor when you receive the NULLEVENT message. Use the Arrow-Cursor, IBeamCursor, HandCursor, WatchCursor, or SpinCursor XOPSupport routines or the Macintosh SetCursor routine to do this. You also need to respond to messages like ACTIVATE, CLICK, KEY, and many other messages. These are listed in the section **Messages for XOPs with Windows** on page 115. The files WindowXOP1.c and WindowXOP1Win.c illustrate how to respond to these messages.

The items in the File and Edit menus may or may not apply to your window. When you receive the MENUENABLE message and your window is the top window you need to set these items appropriately. The section **Enabling and Disabling Menu Items** on page 242 describes how to do this.

## Adding a Simple Window on Windows

The programming that you need to do to add a window to Igor is pretty close to standard Windows programming techniques. You create a window class by calling the Windows RegisterClass function. You then create a window using the Windows CreateWindowEx function. Your window procedure, which you identified to Windows when you created the class, receives messages (e.g., WM_KEY, WM_CHAR, WM_ACTIVATE, WM_PAINT) directly from the Windows OS. The file WindowXOP1Win.c illustrates using these Windows routines and responding to Windows OS messages.

The main difference between Windows XOP programming and standard Windows programming is that you must call the routine SendWinMessageToIgor from your window procedure. This gives Igor a chance to do housekeeping associated with your window. SendWinMessageToIgor is described in detail below.

Another significant difference is that you will not receive WM_COMMAND messages about menu selections through your window procedure. Rather, you will receive MENUITEM messages from Igor through your XOPEntry routine. Similarly, you will receive MENUENABLE messages from Igor instead of WM_INITMENU messages from Windows.

Igor is an MDI (Multiple Document Interface) application. Because of this, all Igor windows, including XOP windows, must be MDI child windows. Other kinds of windows, including overlapped windows and modeless dialogs are not supported because they would require coordination with Igor that is not implemented.

Implementing a window requires that you write at least the following routines. The names used here are from WindowXOP1. You can use any names that you like for your XOP.

| Routine | Explanation |
|---|---|
| CreateXOPWindowClass | Registers a window class with the Windows OS. Called from the main function. |
| CreateXOPWindow | Creates an XOP window using the registered class. |
| DestroyXOPWindow | Destroys the XOP window using the WM_MDIDESTROY message. |
| XOPWindowProc | This is the window procedure to which the Windows OS sends messages and from which you must call the SendWinMessage-ToIgor function. |

Please refer to the WindowXOP1.c and WindowXOP1Win.c files while reading the following discussion of these routines.

## CreateXOPWindowClass

CreateXOPWindowClass is called from the XOP's main function in WindowXOP1.c and is defined in WindowXOP1Win.c. It uses standard Windows programming techniques, calling the Windows RegisterClass function. Notice that the wndclass.hInstance field is set to the XOP's HMODULE, which we get using the XOPModule XOPSupport routine.

If you make your XOP transient, using the SetXOPType XOPSupport call, Igor will send you the QUIT message and will then remove your XOP from memory. The user could later invoke your XOP again, causing Igor to load it into memory again, and causing your CreateXOPWindow-Class routine to run again. Registering a window class that already exists is an error in Windows. The Windows OS documentation does not discuss how to handle this situation. Therefore, we recommend that you not make a Windows XOP that creates a window transient.

## CreateXOPWindow

CreateXOPWindow is called from the WindowXOP1 main function in WindowXOP1.c and is defined in WindowXOP1Win.c. Again, this function uses standard Windows programming techniques. Note that the results from the XOPModule and IgorClientHWND XOPSupport functions are passed to CreateWindowEx.

XOP windows must be MDI child windows. This requirement is satisfied by using the WS_CHILD and WS_EX_MDICHILD constants and by specifying IgorClientHWND as the parent window.

## DestroyXOPWindow

DestroyXOPWindow is called as part of the shutdown process from XOPQuit, after the XOP receives the CLEANUP message from Igor. Igor sends this message to all running XOPs when it is about to quit. DestroyXOPWindow is defined in WindowXOP1Win.c.

You must dispose an MDI child window by sending the WM_MDIDESTROY message to the MDI client window, which you get by calling IgorClientHWND. If you call DestroyWindow rather than sending the WM_MDIDESTROY message, you will leave Igor in an unstable state. After sending the WM_MDIDESTROY message, you will receive a barrage of messages from Windows, including the WM_DESTROY message. It is in response to WM_DESTROY that you should dispose of any storage that you have allocated for the window.

## XOPWindowProc

XOPWindowProc is the window procedure for windows of the class registered by CreateXOP-WindowClass. It is defined in WindowXOP1Win.c.

XOPWindowProc is written using standard Windows programming techniques, except that you must call SendWinMessageToIgor once at the start of the window procedure and once at the end. This gives Igor a chance to do housekeeping associated with your window. For example, when your window is activated, Igor compiles the procedure window if necessary and updates the menu bar, removing any inappropriate menu (e.g., the Graph menu) and installing your XOPs main menu if it has one.

You pass to SendWinMessageToIgor the parameters that your window procedure received, plus a parameter that differentiates the "before" call to SendWinMessageToIgor from the "after" call. If Igor returns non-zero from the before call, then you must ignore the message. For example, if an Igor procedure is running, when you call SendWinMessageToIgor, Igor will return non-zero in response to mouse and keyboard messages. This allows your XOP window to behave like an Igor window, which ignores these events during procedure execution.

Igor does not send the following messages to Windows XOPs, because these kinds of matters are handled through the standard Windows OS messages: ACTIVATE, UPDATE, GROW, WINDOW_MOVED, CLICK, KEY, NULLEVENT. Instead, your window procedure will receive WM_MDIACTIVATE, WM_PAINT, WM_SIZE, WM_MOVE, WM_LBUTTON-DOWN, WM_RBUTTONDOWN, WM_KEY, and WM_CHAR messages from the Windows OS.

Your window procedure will not receive WM_COMMAND messages from Windows when the user chooses your menu items. Instead, you will receive MENUITEM messages from Igor through your XOPEntry routine. This is a consequence of the need to integrate XOP menus and menu items with Igor menus. Your window procedure *will* receive WM_COMMAND messages when the user touches controls in your window.

## Menus in Windows XOP Windows

MDI child windows can not have their own menu bars. For a window to have its own menu bar, it must be an "overlapped" window, not a child window. However, overlapped windows do not fit nicely into MDI applications. Igor does not have the infrastructure needed to compensate for this. Therefore, it is not possible to add overlapped windows to Igor and consequently, it is not possible to have a XOP window that has its own menu bar.

# TU ("Text Utility") Windows

A TU window is a "text utility" window - a simple plain text editing window. The TUDemo and VDT sample XOPs illustrate the use of TU windows.

You create a TU window by calling TUNew2. TUNew2 creates a TU window and returns to the caller a TU handle and an XOP_WINDOW_REF. The XOP passes the TU handle back to Igor to perform various operations on the TU window. The XOP_WINDOW_REF is a WindowPtr on Macintosh and an HWND on Windows.

TUNew2 was added in Igor Pro 3.13 for cross-platform compatibility. The older TUNew function is available to Macintosh XOPs only. New XOPs should use TUNew2 for both Macintosh and Windows.

Igor disposes of a window created via TUNew2 when you call TUDispose.

A Macintosh TU XOP controls the window by handling messages that Igor sends to the XOP's XOPEntry routine and calling the corresponding TU callback routine. For example, when the user chooses Copy from the Edit menu, Igor sends your XOP the COPY message. In response, you call the TUCopy callback function, causing Igor to do the copy.

A Windows TU XOP works the same way except that Igor internally handles certain window-oriented activities without sending a message to the XOP. The window procedure for the TU window is inside Igor. This window procedure handles Windows OS messages like WM_MDIACTIVATE, WM_PAINT, WM_CHAR, and so on, without calling your XOP. Unlike on Macintosh, Igor does not send the ACTIVATE, UPDATE, GROW, WINDOW_MOVED, SETGROW, CLICK, KEY, or NULLEVENT messages to the Windows XOP. Consequently, the Windows XOP has no need to call the corresponding TU callbacks, such as TUActivate, TUUpdate, TUClick, and so on. The handling of all other messages (e.g., COPY) is the same as on Macintosh.

There may be cases in which the Windows TU XOP needs to intervene in the handling of some window-oriented activities. For example, the VDT XOP needs to handle keyboard messages so that it can send characters that the user types to the serial port. To achieve this, the VDT XOP uses subclassing - a technique described in the Windows API documentation.

# Recursion Problems

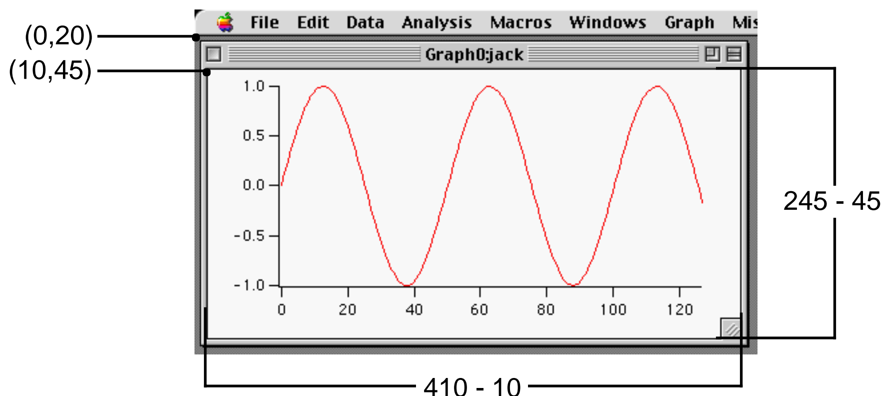
If you add a window to Igor, your XOP may be called recursively. This happens if Igor calls your XOP, your XOP does a callback to Igor, and the callback causes Igor to call your XOP again. For details on when this can happen and how to handle it, see **Handling Recursion** on page 137.

# Igor Window Coordinates

This section describes how Igor stores window sizes and positions in a platform-independent way. You need to know this if you are writing an XOP that creates a window on both Macintosh and Windows and if you want your window's size and position to be roughly maintained when you take an experiment created on one platform and open it on the other.

On the Macintosh, windows are sized and positioned using the SizeWindow and MoveWindow Mac OS routines. These routines use global coordinates in units of pixels, relative to the top/left corner of the main screen. Also, pixels are assumed to be roughly one point square in size and therefore window coordinates can be considered to be in units of points. When specifying a window size and position to the Mac OS, the programmer specifies the size and position of the "content region". This is the area of the window exclusive of the title bar and frame. The following picture shows a graph window created on Macintosh using the command

```
Display/W=(10,45,410,245) jack
```

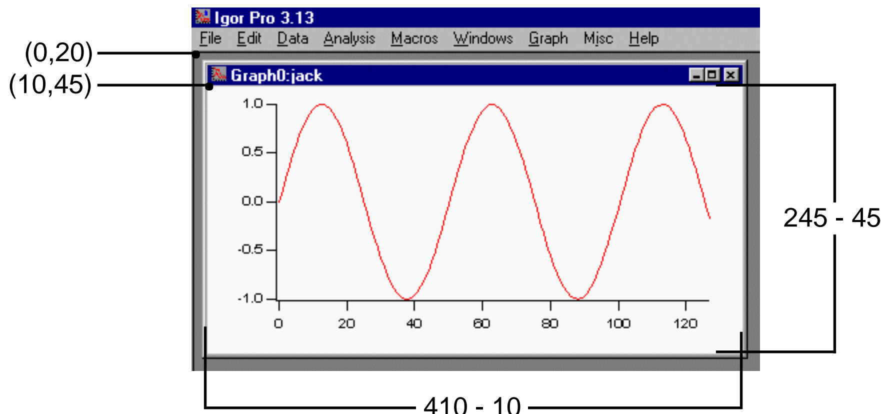



On Windows, windows are sized and positioned using the SetWindowPlacement Win32 routine. This routine uses client coordinates, relative to the top/left corner of the client area of the Igor MDI frame window, in units of pixels. Unlike Macintosh, the relationship between a pixel and a point is variable on Windows. The Windows user can set it by opening the Display control panel, selecting the Settings pane, clicking the Advanced button, and using the Font Size control, which sets the nominal number of pixels per inch.

When WaveMetrics ported Igor Pro to Windows, we needed a way to size and position windows that would work across platforms. For example, if a user took a Macintosh experiment containing the graph shown above to Windows, it should have roughly the same size and position. Moreover, existing experiments and procedures, created long before Igor ran on Windows, had to open and

run correctly on Windows. WaveMetrics defined "Igor Coordinates" to achieve these goals. The following picture shows a graph window created on Windows using the command

```
Display/W=(10,45,410,245) jack
```




Igor coordinates are defined to be in units of points, relative to a spot 20 points above the bottom/left corner of the menu bar. Thus, the coordinates (0, 20) correspond to the bottom/left corner of the menu bar on both platforms. When used to set the size and position of windows, Igor coordinates set the size and position of the content region of the window, not the outside edge of the window frame. We maintain coordinates in floating point because, to accurately reposition a window, we need to use fractional points in /W=(left,top,right,bottom) flags.

Fortunately, XOPSupport routines are available to do all of the necessary conversions between native Macintosh or Windows coordinates and Igor coordinates. To get your window's size and position in Igor coordinates, call the GetXOPWindowIgorPositionAndState XOPSupport routine. If your XOP stores window coordinates, store them using the results returned from this routine. To restore the window to the same size and position, call the SetXOPWindowIgorPosition-AndState XOPSupport routine. These routines work the same on Macintosh and Windows.

## Adding XOP Target Windows

A target window type is a class of windows that can be manipulated using commands from the command line or from Igor procedures. Each target window has a name so that it can be referenced from commands. Igor Pro currently supports five built-in target window types: graphs, tables, page layouts, notebooks, and control panels.

An XOP can add a target window type to Igor. This capability was added to Igor to allow WaveMetrics to better-integrate the Igor Surface Plotter into the main program. An XOP target window behaves substantially like a built-in Igor Pro target window. XOP target windows must be saved as window recreation macros in the Igor Procedure window and in Igor experiment files.

Only very advanced Igor users and XOP programmers should attempt to add a target window type to Igor.

XOPs that create target windows require much more cooperation from Igor than other XOPs. Because they depend on so many Igor Pro behaviors, they are more fragile than normal XOPs. That is, it is more likely that a change in Igor Pro behavior could break this kind of XOP. Because of this fragility and because of the complexity of implementing such an XOP, documentation and support for implementing XOP target windows is not included with the XOP Toolkit. Instead, it is available upon request to registered XOP Toolkit users.

To obtain documentation and a sample XOP for implementing an XOP target window, please send a note to support@wavemetrics.com. Please indicate the version of Igor Pro that you are using, your XOP Toolkit serial number, which development system you are using, the version of the development system, and the application that you have in mind.

*Chapter* 10

# Other Programming Topics


Macintosh Memory Management........ 261
Pointers And Handles........ 262
Using a Handle........ 264
Accessing Igor Data Objects........ 265
Techniques for Cross-Platform Development........ 266
File I/O........ 267
File Path Conversions........ 268
Adding Dialogs........ 269
Alerts and Message Boxes........ 269
Open and Save File Dialogs........ 269
Dialog Resource IDs........ 269
Macintosh Dialogs........ 270
Windows Dialogs........ 270
Cross-Platform Dialogs........ 270
Cross-Platform Dialog Popup Menus........ 275
Macintosh Popup Menus........ 275
Windows Popup Menus........ 276
Creating an Igor-Style Dialog........ 276
Adding Version Resources........ 278
Macintosh Version Resources........ 278
Windows Version Resources........ 278
Structure Alignment........ 279
Shared Structure Alignment........ 279
File Structure Alignment........ 280
Using Igor Structures as Parameters........ 281
Structure Fields........ 282
Strings In Structures........ 283
NVARs and SVARs In Structures........ 283

Calling User-Defined and External Functions ........ 285
Example of Calling a User-Defined or External Function ........ 286
Macintosh Programming Issues ........ 288
Don’t Initialize Macintosh Toolbox Managers ........ 288
Restrictions on Opening Resource Forks ........ 288

# Macintosh Memory Management

Igor uses Macintosh-style memory management, even when running on Windows. Because XOPs need to access Igor data, such as waves, strings, and data folders, XOPs must also use Macintosh-style memory management, at least in these areas. When running on Macintosh, Macintosh memory management routines are provided by the Mac OS. When running on Windows, they are provided by IGOR.lib, with which all Windows XOPs link.

In dealing with your own private data, you can use Macintosh routines, standard C memory management routines, or Windows memory management routines. Using the standard C or Macintosh routines makes it easier to write cross-platform XOPs.

If you are writing a simple XOP, much of the information in this section is for background only. You will not need to manipulate memory directly but instead will call XOPSupport routines. The main thing that you need to know is what a Macintosh handle is, because waves are stored using Macintosh handles. This is discussed later in this section.

The following table lists the Macintosh memory management routines that are commonly used in XOP programming. These routines, as well as some less commonly used routines, are described in detail in the section **Emulated Macintosh Memory Management Routines** on page 497.

| Routine | What It Does |
|---|---|
| void* NewPtr(long size); | Allocates a block of memory in the heap and returns a pointer to it. |
| long GetPtrSize(Ptr p); | Returns the number of bytes in the block of memory pointed to by the pointer. |
| void SetPtrSize(Ptr p, long size); | Sets the number of bytes in the block of memory pointed to by the pointer. |
| void DisposePtr(void* p); | Frees the block of memory pointed to by the pointer. |
| Handle NewHandle(long size); | Allocates a block of memory in the heap and returns a handle to it. |
| long GetHandleSize(Handle h); | Returns the number of bytes in the block of memory referred to by the handle. |
| void SetHandleSize(Handle h, long size); | Sets the number of bytes in the block of memory referred to by the handle. |

*(continued on next page)*

*(continued from previous page)*

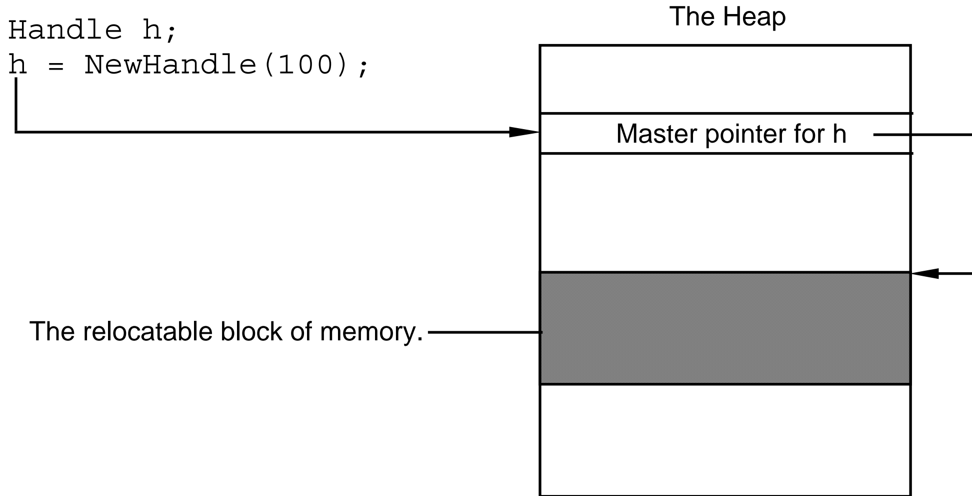
| Routine | What It Does |
|---|---|
| void DisposeHandle(Handle h); | Frees the block of memory pointed to by the handle. |
| int HGetState(Handle h); * | Returns an integer containing bits that specify the state of the block of memory referred to by the handle. |
| void HSetState(Handle h, int state); | Sets the state of the block of memory referred to by the handle. |
| void HLock(Handle h); * | Locks the block of memory referred to by the handle at its current address in the heap. |
| void HUnlock(Handle h); | Unlocks the block of memory referred to by the handle so that it is free to move in the heap. |
| void MoveHHi(Handle h); * | Moves the block of memory referred to by the handle to the top of the heap so that it will not cause heap fragmentation when later locked. |
| int MemError(void); | Returns an error code from the preceding Macintosh memory management call. |

* An XOPSupport routine called MoveLockHandle combines the actions of HGetState, MoveHHi, and HLock, and is usually used instead of these routines.

## Pointers And Handles

When you need to use a large block of memory or if that block needs to persist when the function that creates it returns, you allocate a block of memory in the heap. In XOP programming, you typically use a pointer to a block of memory in the heap if that block will never need to be resized. You use a handle if the block will need to be resized. The function NewPtr allocates a block and returns a pointer to that block while the function NewHandle allocates a block and returns a handle to that block.

A pointer points directly to the block of data in the heap. A handle points indirectly to the block of data - it points to a pointer to the block. The handle contains the address of a master pointer which contains the address of the block of memory in the heap. The following illustration shows the relationship of a handle and the block of memory that it refers to.




A block allocated by NewPtr is called non-relocatable. A block allocated by NewHandle is called relocatable. Using handles allows the memory management system to avoid heap fragmentation by moving relocatable blocks when it needs to allocate a new block.

Because a handle refers to a block of memory indirectly, you must "dereference" the handle before you can access the block. For example, if h is a handle returned by NewHandle, then *h is a pointer to the block of memory that h refers to. By contrast, a pointer returned by NewPtr points directly to the block. The following code snippets illustrates the difference between pointers and handles. Both snippets allocate a block of 2 bytes of memory, set the first byte to the value 1 and the second byte to the value 2, and then dispose the block.

```
Ptr p;
p = NewPtr(2);
if (p == NULL)
   return NOMEM;
*p = 1;
*(p+1) = 2;
DisposePtr(p);
```

```
Handle h;
Ptr p;
h = NewHandle(2);
if (h == NULL)
   return NOMEM;
p = *h;                     // DEREFERENCE
*p = 1;
*(p+1) = 2;
DisposeHandle(h);
```

## Using a Handle

In the second snippet above, once h is dereferenced, p points to a relocatable block of memory in the heap. This means that the memory manager can move the block of memory in order to avoid heap fragmentation. The memory manager will do this only when it allocates a new block of memory or resizes an existing block. If you make a call after dereferencing the handle and if that call directly or indirectly allocates memory, the memory manager may relocate the block, leaving p no longer pointing to the block of memory referenced by h. p is then said to be a "dangling pointer". Because of this, you must be careful about dereferencing handles.

One solution is to lock the block, using HLock, before dereferencing the handle, and to unlock it, using HUnlock, when you are finished accessing the block. There are two problems with this approach. First, locking a block fragments the heap. Second, this approach assumes that the block was unlocked to begin with. However, a calling routine may have already locked it. By unlocking it, we are undermining the calling routine. A solution to these problems is to use MoveLock-Handle and HSetState instead of HLock and HUnlock.

```
Handle h;
Ptr p;
int hState;
h = NewHandle(2);
if (h == NULL)
   return NOMEM;
hState = MoveLockHandle(h);
p = *h;                    // DEREFERENCE
<Do whatever we want with the block>
HSetState(h, hState);
```

MoveLockHandle moves the block of memory to the top of the heap, which prevents heap fragmentation, and then locks it, and then returns the state of the block before it was locked. HSetState restores the block to its original state.

Unfortunately, MoveLockHandle can be slow, because it may cause the memory manager to move a lot of blocks of memory around. A better solution that is often possible is to merely re-dereference the handle when necessary. Here is an example.

```
Handle h;
Ptr p;
h = NewHandle(2);
if (h == NULL)
   return NOMEM;
p = *h;                    // DEREFERENCE
*p = 1;
<Call a subroutine that may allocation memory>
p = *h;                    // RE-DEREFERENCE
*(p+1) = 2;
DisposeHandle(h);
```

This technique is somewhat dangerous. Imagine that the you write the routine without the subroutine call and without the re-dereference. A month later, you add the subroutine call. Will you remember to add the dereference? If you are very careful, this technique is fine.

You can avoid the need to be so careful by making the dereference temporary and eliminating the pointer p.

```
Handle h;
h = NewHandle(2);
if (h == NULL)
   return NOMEM;
**h = 1;
<Call a subroutine that may allocation memory>
*(*h+1) = 2;
DisposeHandle(h);
```

Here, instead of dereferencing h once and storing the dereference, we dereference it each time we need to access the block of memory. In most cases, this is the wise way to do it.

The section **Dangling Pointer / Heap Scramble Problems** on page 312 contains more discussion of handles and dereferencing techiques.

## Accessing Igor Data Objects

You might use NewPtr and DisposePtr or NewHandle and DisposeHandle when dealing with your own private data. When dealing with Igor objects, such as waves, you never allocate and dispose blocks directly. Instead, you call XOPSupport routines, such as MakeWave and KillWave. Also, you don't dereference a wave handle directly. Instead, you call WaveData, which dereferences it for you.

See Chapter 7 for details on accessing Igor data.

## Techniques for Cross-Platform Development

It is best to maintain one set of source code for both the Macintosh and Windows versions of your XOP. Much of the source code, such as routines for number crunching or interfacing with Igor, will be completely platform-independent. Menu-related and file-related routines can also be platform-independent because of the support provided by the XOPSupport library.

Most XOPs can be written in a mostly platform-independent way. The GBLoadWave, MenuXOP1, NIGPIB2, SimpleFit, SimpleLoadWave, TUDemo, WaveAccess, XFUNC1, XFUNC2, XFUNC3, XOP1, and XOP2 sample XOPs have no platform-specific C source files and few platform-related ifdefs. Routines for number crunching or interfacing with Igor are inherently platform-independent. Dialog-related, menu-related and file-related routines can also be written in a mostly platform-independent manner using support provided by the XOPSupport library.

Routines for creating windows, handling user interaction with windows, and drawing in windows will be mostly platform-dependent. Platform-dependent functionality can be handled by conditional compilation within a common source file or by creating parallel source files for each platforms. The WindowXOP1 sample XOP illustrates both of these techniques. WindowXOP1.c has a few platform-dependent ifdefs. The bulk of the platform-dependent code is in WindowXOP1Mac.c and WindowXOP1Win.c. The public routines provided by these files have the same interface regardless of platform. This relieves the main file, WindowXOP1.c, from platform-dependency, for the most part.

# File I/O

The XOP Toolkit provides the following platform-independent routines for creating, deleting, reading, writing, and otherwise dealing with files:

| | | |
|---|---|---|
| XOPFileExists | XOPCreateFile | XOPDeleteFile |
| XOPOpenFile | XOPCloseFile | |
| XOPReadFile | XOPReadFile2 | XOPReadLine |
| XOPWriteFile | | |
| XOPGetFilePosition | XOPSetFilePosition | |
| XOPAtEndOfFile | XOPNumberOfBytesInFile | |

These routines are defined in XOPFiles.c. Some of them (e.g., XOPOpenFile) use full paths to identify files. The full paths must be "native". That is, on Macintosh, they must use colons as path separators and on Windows they must use backslashes. Do not use POSIX paths (with forward slashes), even on Mac OS X, because the Carbon routines called by the XOPSupport routines do not support forward slashes.

The XOP Toolkit provides symbols to use when allocating buffers for volume names, folder names, file names and paths. For example:

```
char volumeName[MAX_VOLUMENAME_LEN+1];
char volumeName[MAX_DIRNAME_LEN+1];
char volumeName[MAX_FILENAME_LEN+1];
char volumeName[MAX_PATH_LEN+1];
```

These statements allocate the appropriate size buffers on all platforms. The "+1" adds room for the null terminator.

The XOP Toolkit also provides the XOPOpenFileDialog and XOPSaveFileDialog routines which allow the user to choose a file to open or to specify where to save a file. Other XOPSupport routines provide the means to obtain a full path to a file based on parameters from a command-line command. The file-loader XOPs SimpleLoadWave and GBLoadWave illustrate the use of these routines.

Earlier XOP Toolkits used Macintosh volume reference numbers, directory IDs, and working directory reference numbers. These things are platform-dependent and are no longer used by the sample XOPs. As of version 3.1, the XOP Toolkit uses full paths in place of volume reference numbers, directory IDs, and working directory reference numbers.

## File Path Conversions

The Macintosh and Windows operating systems use different syntax for creating a path to a file. Here are some examples.

| Macintosh | Windows |
|---|---|
| hd:Folder1:Folder2:File | C:\Folder1\Folder2\File |
| :Folder2:File | .\Folder2\File |

On Mac OS X, POSIX paths with forward slashes are sometimes used. However, Igor and the XOP Toolkit are based on Apple's Carbon API which does not use POSIX paths. Carbon uses traditional Macintosh paths with colon separators. When we speak of "Macintosh paths", we are talking about colon-separated paths.

Igor commands can refer to files using paths. So that an Igor programmer can write procedures that work on any platform, commands must work regardless of which platform they are executing on. For example, the commands:

```
LoadWave/J/P=Igor ":Examples:Programming:AutoGraph Data:Data0"
LoadWave/J/P=Igor ".\\Examples\\Programming\\AutoGraph Data\\Data0"
```

must work on both Macintosh and Windows. (Note that in a literal string in an Igor command, you must use "\\" to represent a single backslash because backslash is an escape character in Igor.}

This requirement means that an XOP that deals with a file path must accept the path in either Macintosh or Windows format. The technique used by the sample XOPs is to convert all paths to the native format as soon as the path is received from the command. "Native" means that the path uses Macintosh syntax when running on Macintosh and Windows syntax when running on Windows. For example, SimpleLoadWaveOperation.c calls the GetNativePath routine to convert its input parameter.

GetNativePath is an XOPSupport routine that converts a path to the conventions of the platform on which the XOP is running. On Macintosh, it converts to Macintosh conventions and on Windows, it converts to Windows conventions. If the path already uses the correct conventions, GetNativePath does nothing. GetNativePath calls other XOPSupport routines, MacToWinPath and WinToMacPath. If you use the technique of converting all paths to native format as soon as possible, then you will not need to use MacToWinPath and WinToMacPath directly but can use just c.

Other cross-platform path-related XOPSupport routines include ConcatenatePaths, GetDirectoryAndFileNameFromFullPath, FullPathPointsToFile, and FullPathPointsToFolder.

# Adding Dialogs

When the user selects your XOP's menu item you may want to put up a dialog. The XOP Toolkit includes support for implementing dialogs on both Macintosh and Windows. This section describes that support.

You do not need to use the XOP Toolkit dialog routines to implement a dialog. You can use any valid technique. However, using the XOPSupport routines and the techniques illustrated by the sample XOPs provides you with a way to create one set of dialog code that runs on both Macintosh and Windows.

## Alerts and Message Boxes

The following routines provide a cross-platform way to display simple alert dialogs.

| Routine | What It Does |
|---|---|
| XOPOKAlert | Displays a message with an OK button. |
| XOPOKCancelAlert | Displays a message with OK and Cancel buttons. |
| XOPYesNoAlert | Displays a message with Yes and No buttons. |
| XOPYesNoCancelAlert | Displays a message with Yes, No, and Cancel buttons. |

## Open and Save File Dialogs

These routines provide a cross-platform way to implement Open File and Save File dialogs.

| Routine | What It Does |
|---|---|
| XOPOpenFileDialog | Displays a standard Open File dialog. |
| XOPSaveFileDialog | Displays a standard Save File dialog. |

## Dialog Resource IDs

By convention, Macintosh XOP resource IDs for DLOG and DITL resources start from 1256. ID 1256 is reserved for an open file dialog and ID 1257 is reserved for a save file dialog, if these are needed by the XOP. Use ID numbers 1258 through 1299 for other XOP dialogs. The range 1100 through 1199 is also available for use in XOPs.

On Windows, you will typically let the Windows resource editor assign dialog resource ID numbers. The Visual C++ or CodeWarrior development systems store the resource ID numbers in the resource.h file associated with each project.

## Macintosh Dialogs

Implementing a modal dialog in a Macintosh XOP is the same as in a standalone program with a few exceptions.

An XOP must use the XOPSupport routines GetNewXOPDialog, DoXOPDialog, and DisposeXOPDialog instead of the corresponding Macintosh routines GetNewDialog, ModalDialog, and DisposeDialog.

As of Carbon, XOPs must use Apple Appearance Manager-compatible techniques to create dialogs. This includes creating a dlgx resource with the kDialogFlagsUseControlHierarchy bit set. See Apple's Appearance Manager documentation and the sample XOPs for details.

Creating the Macintosh resources for an XOP dialog presents a bit of a problem. Apple no longer provides tools for creating dialog resources. You must either edit the .r file in a text editor or use a now-obsolete resource editor such as ResEdit. For more on Macintosh resource creation, see **Creating Resources on Macintosh** on page 109.

You may prefer to use more up-to-date, Mac OS X-savvy techniques for creating dialogs, using nibs and Interface Builder. The XOP Toolkit provides no support for this but there is no reason why you can't do it in your own code.

## Windows Dialogs

Implementing a modal dialog in a Windows XOP is the same as in a standalone program. You create the dialog resources using the Visual C++ resource editor. For more on Windows resource creation, see **Creating Resources on Windows** on page 109.

## Cross-Platform Dialogs

In the sample XOPs (e.g., GBLoadWave), dialogs are implemented using about 90% platform-independent code and 10% platform-dependent code. The platform-independent code relies on XOP Toolkit dialog utilities (e.g., SetCheckBox, GetDText, CreatePopMenu) that are implemented for both platforms. These utility routines take parameters of type XOP_DIALOG_REF. An XOP_DIALOG_REF is a DialogPtr on Macintosh and an HWND on Windows.

The lower level code, for example the code that responds to hits on checkboxes and buttons, is platform-independent. The highest level dialog code, for example the code that creates the dialog window, is platform-dependent. This arises from the fact that the Macintosh and Windows methods of implementing dialogs have very different flows.

Here is an outline of a typical Macintosh dialog routine in a standalone program.

```
int
MacintoshDialog(<parameters>)
{
   <local variables>

   GetNewDialog
   Initialize dialog items using function input parameters
   do
      ModalDialog
      switch(itemHit) {
         Handle hits on buttons, checkboxes, etc.
      }
   until done
   Return dialog results using function output parameters
   DisposeDialog
   return result
}
```

And here is a typical Windows dialog routine in a standalone program.

```
<global variables>

DialogProc(HWND hwnd, UINT msgCode, WPARAM wParam, LPARAM lParam)
{
   switch(msgCode) {
      case WM_INITDIALOG:
         Initialize dialog items using globals.
         break;
      case WM_COMMAND:
         Handle hits on buttons, checkboxes, etc.
         break;
   }
}

int
WindowsDialog(void)
{
   Set global variables using globals
   DialogBox(DialogProc);
   Return dialog results using globals
   return result
}
```

On Macintosh, the programmer retains control in a do-loop. On Windows, the operating system retains control, sending messages to the dialog procedure as needed.

On Macintosh, we normally use local variables to store values used during dialog execution. The local variables are defined in the highest level dialog routine - the one that creates the dialog, loops until the user clicks OK or Cancel, and then disposes the dialog. This does not work on Windows because the loop is inside the Windows OS (in the DialogBox or DialogBoxParam function), not in our program. Therefore, there is no place to store local variables.

To allow the lower-level, platform-independent routines to access dialog-related variables on both platforms, we encapsulate these variables in a structure. In the GBLoadWaveDialog.c file, this structure is called a DialogStorage structure. The structure stores all of the values needed to keep track of things while the dialog is running. On both platforms, the structure is stored as a local variable in the highest level dialog routine, GBLoadWaveDialog in GBLoadWaveDialog.c. This highest level routine is implmented separately for each platform, using ifdefs. On Windows, the dialog procedure callback function gains access to this structure via the lParam that comes with the WM_INITDIALOG message.

Also to allow the lower-level routines to be platform-independent, we arrange things so that the Windows dialog item IDs match the Macintosh dialog item numbers. Macintosh dialog item numbers start from 1 and increment sequentially. To match this, when we define our Windows dialog resource, we use the same numbering scheme. This allows the low-level common routines to refer to dialog items using the same symbol regardless of platform.

By using XOP Toolkit cross-platform dialog utility routines, matching Windows dialog item IDs to Macintosh dialog item numbers, and using the DialogStorage structure, we are able to write dialog code that runs on both platforms without much platform-specific code. In the following outlines, the underlined routines are identical for Macintosh and Windows.

Here is an outline of the resulting Macintosh GBLoadWave dialog routine.

```
int
GBLoadWaveDialog(void)     // Macintosh
{
   DialogStorage ds;

   InitDialogStorage(&ds);
   theDialog = GetXOPDialog(DIALOG_TEMPLATE_ID);
   InitDialogSettings(theDialog, &ds);

   do {

      DoXOPDialog(&itemHit);
      switch(itemHit) {
         HandleItemHit(theDialog, itemHit, &ds);
      }
      ShowCmd(theDialog, &ds, cmd);
   } while (itemHit<DOIT_BUTTON || itemHit>TOCLIP_BUTTON);

   ShutdownDialogSettings(theDialog, &ds);
   DisposeDialogStorage(&ds);

   DisposeXOPDialog(theDialog);

   return itemHit==CANCEL_BUTTON ? -1 : 0;
}
```

Here is an outline of the resulting Windows GBLoadWave dialog routine.

```
static BOOL CALLBACK
DialogProc(HWND theDialog, UINT msgCode, WPARAM wParam, LPARAM lParam)
{
   static DialogStoragePtr dsp;

   itemID = LOWORD(wParam);   // Identifies the item hit.
   switch(msgCode) {
      case WM_INITDIALOG:
         dsp = (DialogStoragePtr)lParam;
         InitDialogSettings(theDialog, dsp);
         break;

      case WM_COMMAND:
         switch(itemID ) {
            case DOIT_BUTTON:
            case TOCMD_BUTTON:
            case TOCLIP_BUTTON:
            case CANCEL_BUTTON:
               HandleItemHit(theDialog, itemID, dsp);
               ShutdownDialogSettings(theDialog, dsp);
               EndDialog(theDialog, itemID);
               break;
            default:
               HandleItemHit(theDialog, itemID, dsp);
               ShowCmd(theDialog, dsp, cmd);
               break;
         }
         break;
   }
}

int
GBLoadWaveDialog(void)     // Windows
{
   DialogStorage ds;
   int result;

   if (result = InitDialogStorage(&ds))
      return result;

   result = DialogBoxParam(. . . , DialogProc, (LPARAM)&ds);

   DisposeDialogStorage(&ds);

   if (result != CANCEL_BUTTON)
      return 0;

   return -1;              // Cancel.
}
```

The code in VDTDialog.c uses the same structure and the same platform-independent routines.

## Cross-Platform Dialog Popup Menus

The XOP Toolkit provides cross-platform dialog popup menu support. On Macintosh, popup menus is implemented using the Macintosh menus. On Windows, popup menus are implemented using combo boxes.

The GBLoadWaveDialog.c file illustrates how to use dialog popup menu XOPSupport routines. The InitDialogSettings routine calls InitPopMenus. You must call InitPopMenus when you initialize a dialog, before calling any other dialog popup menu routines.

Next, InitDialogSettings calls InitDialogPopups. InitDialogPopups calls CreatePopMenu once for each popup menu in the dialog. On Macintosh, CreatePopMenu creates a new menu. On all platforms, CreatePopMenu sets the initial contents of the popup menu and sets the initial selection.

The HandleItemHit routine shows how to respond to a click on a popup menu. HandleItemHit calls GetPopMenu to get the new selection. Other parts of the code also call GetPopMenu whenever they need to know what is selected.

When the user dismisses the dialog, the ShutdownDialogSettings routine calls KillPopMenus. This balances the InitPopMenus routine. Also, on Macintosh, it disposes the menus created by CreatePopMenu. After calling KillPopMenus, you must not call any further dialog popup menu support routines.

### Macintosh Popup Menus

On Macintosh a dialog popup menu item is defined as a Control in the DITL resource. There must be a corresponding CNTL resource. Your CNTL resources should use resource IDs in the range 1100 to 1199. Make sure that the bounds rectangle in the CNTL resource matches the bounds rectangle for the corresponding dialog item in the DITL resource.

The CNTL resource fields are nominally called initial value, visibility, maximum value and minimum value. However, when used for a popup menu, they really mean something else. The initial value field really stores something called the "title constant". The maximum field really stores the width of the title in pixels. We use 0 for these because we create an explicit title item. The minimum field really stores a MENU resource ID. This kludge is more or less explained in Apple's Control Manager documentation.

You must specify -12345 as the MENU resource ID. This prevents the Mac OS Control Manager from attempting to create a menu from a resource. The menu is created when you call the CreatePopMenu XOPSupport routine and is disposed when you call DisposeXOPDialog.

Make sure to set the bounds field of the CNTL resource to the same coordinates as the corresponding item in the DITL.

### Windows Popup Menus

The XOP Toolkit implements popup menus on Windows using combo boxes. Combo boxes do not support disabling of items. In applications where you would normally disable a popup menu item on Macintosh, to indicate that the item is not consistent with other dialog selections, you must find an alternative approach. One approach is to allow the user to select an illegal item and then to clearly indicate that the selection is illegal. Another approach is to remove the item from the popup menu when it is not available and to add it back when it is available.

For a complete list of dialog popup menu support routines, see **Dialog Popup Menus** on page 405.

## Creating an Igor-Style Dialog

If your XOP adds a command line operation, you may want to add an Igor-style dialog. In most cases, adding the dialog will be harder than adding the operation itself.

An Igor-style dialog has certain standard controls, namely the Do It, To Cmd, To Clip, Help and Cancel buttons. It may also have lists of waves or other objects, popup menus and standard controls like text items, radio buttons and checkboxes. In an Igor-style dialog, when the user types and clicks, the dialog generates a command which is displayed in a command box as it is generated.

The GBLoadWave sample XOP implements a fairly elaborate Igor-style dialog and can serve as a starting point.

The basic structure of the code to create an Igor-style dialog is as follows:

```
Create a dialog window
Preset all of the controls
Try to restore the controls to their previous state

do
   Find what control was touched and respond appropriately
   Generate and display the command based on state of controls
until (Do It or To Cmd or To Clip or Cancel)

if (not Cancel)
   Put the command in Igor's command buffer

Save the state of the dialog controls
Dispose dialog window
```

The GBLoadWave sample XOP shows how to implement each of these steps, including saving and restoring the dialog's settings and creating and displaying the command generated by the dialog.

In Igor-style dialogs, the first seven items are standard:

```
#define DOIT_BUTTON 1
#define CANCEL_BUTTON 2
#define TOCMD_BUTTON 3
#define TOCLIP_BUTTON 4
#define CMD_BOX 5
#define HELP_BUTTON 6
#define TITLE 7
```

As the user types and clicks in the dialog, the XOP displays the command being generated in the command box, using the DisplayDialogCmd XOPSupport routine

When the user clicks Do It, To Cmd, or To Clip, the XOP calls the FinishDialogCmd XOPSupport routine, which does the appropriate thing with the command generated by the dialog.

When the user clicks the Help button, the dialog calls XOPDisplayHelpTopic. XOPDisplayHelp-Topic displays the specified topic in the Igor help file associated with the XOP. For details on creating an Igor help file, see Chapter 11.

The title item is used on Macintosh but not on Windows, where the title appears in the window caption. It is also possible to display the title in the Macintosh window frame rather than as a dialog item if you use the right dialog procedure ID in your DLOG resource (movableDBoxProc instead of dBoxProc).

The XOPSupport routines include extensive support for popup menus in dialogs. The GBLoadWaveDialog.c file illustrates shows how to use a popup menu to display a list of Igor symbolic paths.

# Adding Version Resources

If you plan to distribute your XOP to other people, it is a good idea to add version resources. The only use for the version resources is to identify your XOP to a person doing a Get Info in the Macintosh Finder or viewing properties in the Windows desktop. All of the sample XOPs have version resources.

## Macintosh Version Resources

You create Macintosh version resources by editing your XOP's .r file or using ResEdit. The ‘vers, 1’ resource identifies your XOP’s version number. You can use any version number that you like. The ‘vers, 2’ resource identifies the version of Igor with which your XOP is intended to run.

Here is an example:

```
resource 'vers' (1) {                  // Version of XOP
   0x01, 0x00, release, 0x00, 0,       // Version bytes and country code
   "1.00",
   "1.00, © 2004 WaveMetrics, Inc., all rights reserved."
};

resource 'vers' (2) {                  // Version of Igor
   0x05, 0x00, release, 0x00, 0,       // Version bytes and country code
   "5.00",
   "(for Igor Pro 5.00 or later)"
};
```

## Windows Version Resources

You create a version resource using the Visual C++ or CodeWarrior resource editor. The editor stores the resource in your XOP's main .rc file.

# Structure Alignment

Structure alignment controls whether fields in structures are aligned on two, four or eight-byte boundaries and whether padding between fields is used to achieve this alignment. The C and C++ languages do not define structure alignment so it is compiler-dependent.

There are two ways to set structure alignment. First, you can use a project setting to set the project-default structure alignment. Second, you can use pragma statements when defining specific structures to override the default alignment.

## Shared Structure Alignment

Structure alignment is critical when an executable passes a structure to a separately-compiled executable, such as when Igor calls an XOP or vice versa. If they don't agree on structure alignment, they can not correctly access structure fields. This can cause bizarre crashes that are difficult to debug.

Agreement is needed only for structures passed between separately-compiled executables, such as Igor and an XOP. We call these "shared" structures.

In the XOP Toolkit, shared structures are defined with two-byte alignment. This includes structures defined in XOPSupport header files as well as structures defined in the XOP itself, namely the structures used to pass parameters to an XOP's external operations and functions. Two-byte alignment is a legacy from Igor's early days when it was the Macintosh standard alignment.

XOPSupport structures are defined in XOPSupport header files such as XOP.h and IgorXOP.h. These files contain statements to insure two-byte alignment. For example:

```
#include "XOPStructureAlignmentTwoByte.h"

<Structures defined here>

#include "XOPStructureAlignmentReset.h"
```

The #included files contain the pragma statement to set the structure alignment. The necessary pragma statements are different for different compilers. The included files take care of those compiler differences.

In addition to the shared structures defined in XOPSupport header files, individual XOPs must also define shared structures for external operation and function parameters which they receive from Igor. Such shared parameter structures must be two-byte aligned and so must use the

#include statements shown above. Thus you will see these statements in all of the sample XOP projects.

Failure to use two-byte alignment for shared structures will cause crashes that are sometimes difficult to diagnose. Therefore, although the use of pragma statements is sufficient if they are always used when needed, in the sample CodeWarrior and Visual C++ XOPs, we have usually specified two-byte alignment via the project settings dialogs. This tells the compiler to use two-byte alignment by default and saves the XOP programmer who forgets to use the pragma statements.We did not do this in the sample Xcode projects because Xcode does not provide a user-interface for the structure alignment setting.

If you are an advanced programmer you may want to set the project default differently. To make sure that this is safe, follow these steps:

1. Verify that any external parameter block structures in your XOP are set to two-byte alignment using #include statements.
2. Use the project settings dialog to set the default structure alignment to the normal setting (usually 8 bytes).
3. Recompile your XOP and test it.

## File Structure Alignment

You also need to be careful about structure alignment if you store a structure in a file. Once you define the file structure, you must use the same structure and alignment in all future versions of your XOP.

If your XOP is to run cross-platform, you must use the same structure and alignment on all platforms.

WaveMetrics uses two-byte alignment for structures stored on disk.

# Using Igor Structures as Parameters

Igor Pro 5.03 added the ability to pass a pointer to a structure as a parameter to an external operation or external function. This is a technique for advanced programmers.

If you use this feature, your XOP will require Igor Pro 5.03 or later. You should put a test in your main function to make sure that you are running with a recent enough version. See **Checking Igor's Version** on page 140 for details.

Structure parameters are passed as pointers to structures. These pointers always belong to Igor. You must never dispose or resize a structure pointer but you may read and write its fields.

An instance of an Igor structure can be created only in a user-defined function and exists only while that function is running. Therefore, when a structure must be passed to an external operation or function, the operation or function must be called from a user-defined function, not from the command line or from a macro. An external operation that has an optional structure parameter can be called from the command line or from a macro if the optional structure parameter is not used.

The pointer for a structure parameter can be NULL. This would happen in an external operation or function if the user supplies * as the parameter or in the event of an internal error in Igor. Therefore you must always test a structure parameter to make sure it is non-NULL before using it.

If you receive a NULL structure pointer as a parameter and the parameter is required, return an EXPECTED_STRUCT error code. Otherwise, interpret this to mean that the user wants default behavior.

You must make sure that the definition of the structure in Igor and in the XOP are consistent. Otherwise a crash is likely to occur.

For examples, see **External Operation Structure Parameter Example** on page 169 and **External Function Structure Parameter Example** on page 196.

## Structure Fields

This table shows the correspondence between Igor structure field types and C structure field types.

| Igor Field Type | C Field Type | Notes |
|---|---|---|
| Variable | double | |
| Variable/C | double[2] | |
| String | Handle | See **Strings In Structures** on page 283. |
| WAVE | waveHndl | Always check for NULL. |
| NVAR | NVARRec | Use with GetNVAR and SetNVAR. |
| SVAR | SVARRec | Use with GetSVAR and SetSVAR. |
| FUNCREF | void* | Use with GetFunctionInfoFromFuncRef. |
| STRUCT | struct | Embedded substructure. |
| char | char | |
| uchar | unsigned char | |
| int16 | short | |
| uint16 | unsigned short | |
| int32 | long | |
| uint32 | unsigned long | |
| float | float | |
| double | double | |

If the calling Igor procedure attempts to access a non-existent wave, the corresponding waveHndl structure field will be NULL. Thus the external function must always check for NULL before using a waveHndl field.

If the calling Igor procedure attempts to access a non-existent global numeric variable or global string variable, the GetNVAR, SetNVAR, GetSVAR and SetSVAR XOPSupport routines will return an appropriate error code.

The FUNCREF field type can be used to call an Igor user-defined function or an external function that is referenced by a field in an Igor Pro structure. See **Calling User-Defined and External Functions** on page 285 for details.

## Strings In Structures

Strings in Igor structures behave just like string parameters passed directly to an external operation implemented with Operation Handler. They also behave like string parametes passed directly to an external function with one important exception. In the case of a string passed as a simple external function string parameter, the XOP owns the string handle and must dispose it. In the case of a string field in a structure, Igor owns the handle and the external function *must not dispose it*.

A string field handle can be NULL. The XOP must check for NULL before using the field.

As with simple string parameters, strings referenced by structure fields are stored in plain Macintosh-style handles, even when running on Windows. The handle contains the string's text, with neither a count byte nor a trailing null byte. Use GetHandleSize to find the number of characters in the string. To use C string functions on this text you need to copy it to a local buffer and null-terminate (using GetCStringFromHandle) it or add a null terminator to the handle and lock the handle. In the later case, you must remove the null terminator and unlock the handle when you are finished using it as a C string.

## NVARs and SVARs In Structures

This example illustrates handling Igor structures containing NVARs, which reference global numeric variables, and  SVARs which reference global string variables.

```
#include "XOPStructureAlignmentTwoByte.h" // Set struct alignment

#define kF2StructureVersion 1000    // 1000 means 1.000.
struct F2Struct {               // Format of structure parameter.
   unsigned long version;
   NVARRec nv;                  // Corresponds to NVAR field in Igor structure.
   SVARRec sv;                  // Corresponds to SVAR field in Igor structure.
};
typedef struct F2Struct F2Struct;

struct F2Param {                // Parameter structure.
   F2Struct* sp;
   double result;
};
typedef struct F2Param F2Param;

#include "XOPStructureAlignmentReset.h"
```

```
int
XTestF2Struct(struct F2Param* p)
{
   struct F2Struct* sp;
   NVARRec* nvp;
   double realPart, imagPart;
   SVARRec* svp;
   Handle igorStrH, ourStrH;
   char buffer[256];
   int numType, err=0;

   ourStrH = NULL;
   sp = p->sp;
   if (sp == NULL)  {
      err = EXPECT_STRUCT;
      goto done;
   }
   if (sp->version != 1000) {
      err = INCOMPATIBLE_STRUCT_VERSION;
      goto done;
   }

   nvp = &sp->nv;                  // Handle the NVAR.
   if (err = GetNVAR(nvp, &realPart, &imagPart, &numType))
      goto done;                   // Probably referencing non-existent global.
   realPart *= 2; imagPart *= 2;
   if (err = SetNVAR(nvp, &realPart, &imagPart))
      goto done;

   svp = &sp->sv;                  // Handle the SVAR.
   if (err = GetSVAR(svp, &igorStrH))  // igorStrH can be NULL. Igor owns it.
      goto done;                   // Probably referencing non-existent global.
   if (err = GetCStringFromHandle(igorStrH, buffer, sizeof(buffer)-1))
      goto done;                   // String too long.
   ourStrH = NewHandle(0L);        // We own this handle.
   if (ourStrH == NULL) {
      err = NOMEM;
      goto done;
   }
   if (err = PutCStringInHandle("Hello", ourStrH))
      goto done;
   if (err = SetSVAR(svp, ourStrH))
      goto done;

done:
   if (ourStrH != NULL)
      DisposeHandle(ourStrH);
   p->result = err;
   return err;
}
```

# Calling User-Defined and External Functions

As of Igor Pro 5, an XOP can call an Igor Pro user-defined function or an external function defined in another XOP. You might want to do this, for example, to implement your own user-defined curve fitting algorithm. This is an advanced feature that most XOP programmers will not need.

There are two ways to identify the function to be called: by the function name or using a FUNCREF field in an Igor Pro structure. The ability to call a function by name was added in Igor Pro 5.00. The ability to call a function referenced by a FUNCREF field was added in Igor Pro 5.03. If you use these features, you must check the version of Igor with which you are running. See **Checking Igor's Version** on page 140 for details.

From an XOP's point of view, an Igor user-defined function and an external function defined in another XOP appear to be the same.

There are several difficulties involved in calling a user-defined function:

- You must make sure Igor's procedures are in a compiled state.
- You need some way to refer to the function that Igor and your XOP agree upon.
- You must make sure that the function's parameters and return type are appropriate for your XOP's purposes.

The XOPSupport GetFunctionInfo, GetFunctionInfoFromFuncRef, CheckFunctionForm and CallFunction routines work together with your XOP to address these difficulties. The details of each of these routines are described in Chapter 13.

GetFunctionInfo takes a function name, which you might have received as a parameter to your external operation, and returns information about the function's compilation state, its parameters and its return type. At this time you can call CheckFunctionForm to make sure that the function is appropriate for your purposes.

GetFunctionInfoFromFuncRef works the same as GetFunctionInfo except that, instead of passing the name of a function, you pass the contents of a FUNCREF field in an Igor Pro structure that you have received as a parameter.

Once you have obtained the function information, the rest of the process is the same, whether you used GetFunctionInfo or GetFunctionInfoFromFuncRef.

Since the function may be recompiled or deleted at any time, you must call CheckFunctionForm again shortly before you attempt to call the function.

Once you have successfully called GetFunctionInfo or GetFunctionInfoFromFuncRef and CheckFunctionForm, you can call CallFunction to call the function.

## Example of Calling a User-Defined or External Function

In this example, we have written our own curve fitting routine, analogous to Igor's FuncFit operation, as an external operation. We want to call a user or external function from our external operation.

The function that we want to call has this form:

```
   Function FitFunc(w, x)
      Wave w
      Variable x
```

To simplify the example, we assume that we known the name of the function that we want to execute.

```
// Define the parameter structure.
// These are parameters we will pass to user or external function.

#include "XOPStructureAlignmentTwoByte.h" // Set structure alignment.
struct OurParams {              // Used to pass parameters to the function.
   waveHndl waveH;              // For the first function parameter.
   double x;                    // For the second function parameter.
};
typedef struct OurParams OurParams;
typedef struct OurParams* OurParamsPtr;

#include "XOPStructureAlignmentReset.h"   // Reset structure alignment.
```

```
int
DoOurOperation(waveHndl coefsWaveH)
{
   FunctionInfo fi;
   OurParams parameters;
   int badParameterNumber;
   int requiredParameterTypes[2];
   double result;
   int i;
   double values[5];
   int err;

   // Make sure the function exists and get information about it.
   if (err = GetFunctionInfo("TestFitFunc", &fi))
      return err;

   // Make sure the function has the right form.
   requiredParameterTypes[0]=NT_FP64;  // First parameter is numeric
   requiredParameterTypes[1]=WAVE_TYPE;// Second parameter is a numeric wave.
   if (err = CheckFunctionForm(&fi, 2, requiredParameterTypes,
                                          &badParameterNumber, NT_FP64))
      return err;

   // We have a valid function. Let's call it.

   parameters.x = 0;
   parameters.waveH = coefsWaveH;
   for(i=0; i<5; i+=1) {
      parameters.x = i;
      if (err = CallFunction(&fi, (void*)&parameters, &result))
         return err;
      values[i] = result;
   }

   return 0;
}
```

# Macintosh Programming Issues

This section discusses certain issues that arise when programming an XOP for Macintosh. The vast majority of XOP programmers will not run into these issues.

## Don't Initialize Macintosh Toolbox Managers

An XOP must not initialize any Macintosh Toolbox managers. Igor initializes them and, as far as the toolbox is concerned, the XOP is part of Igor.

## Restrictions on Opening Resource Forks

This section discusses a problem that does not affect most XOPs. It does affect XOPs that directly or indirectly (through system or library calls) access resources.

Prior to Carbon, Igor did some sleight-of-hand to make sure that the resources in all XOPs were hidden from Igor and from all other XOPs. With Carbon, the trick is no longer feasible. All XOP resource forks are now visible to the Resource Manager all of the time. This has the potential to cause problems.

Before Igor sends a message to your XOP's XOPEntry routine, it sets the current resource fork to your XOP's resource fork. However, for speed reasons, it does not do this when calling a direct external operations and function. Also, you may use Macintosh programming techniques, such as Carbon Events, which result in your code being called by the operating system, not by Igor. In this case, you can not be sure what resource fork is current. This means that you must take great care when accessing resources to make sure that you access only your own resources. Here are guidelines for doing this:

1. Before calling any function that directly or indirectly accesses resources, call UseResFile to make sure that the file in which the resource resides is the current resource file. When you are finished, set the current resource file back to what it was. For example:

   ```
   int saveResFile = CurResFile();
   UseResFile(XOPRefNum());          // Set current to XOP resource fork.
   <Access resources>
   UseResFile(saveResFile);
   ```

2. Don't call routines that search multiple resource forks. For example, use Get1Resource, not GetResource. Use Get1NamedResource, not GetNamedResource.
3. There are some Mac OS routines which do not give you the option of restricting the search to one resource fork. These include GetMenu and GetIndString. This does not cause a problem if you set the current resource fork before calling these routines and if the resource is found. However, if the resource is absent from the resource fork that you intend to search, these routines might find the resource in another resource fork.

Chapter 11

# Providing Help

Overview ........ 291
Igor Pro Help File ........ 292
Help for External Operations and Functions ........ 292
Macintosh Balloon Help ........ 295
Balloon Help For Macintosh XOP Menu Items ........ 296
Status Line Help For Windows XOP Menu Items ........ 297
    Status Line Help For Items Added To Igor Menus ........ 297
    Status Line Help For XOP Menus ........ 297
Context-Sensitive Help For Windows XOP Dialogs ........ 300
Help For XOP Dialogs and Windows ........ 300

# Overview

If your XOP will be used by other people, you should provide help. This chapter discusses providing help in one of the following ways:

- Igor Pro help file
- Macintosh balloon help
- Windows status line help
- Windows context sensitive help

The most important of these is the Igor Pro help file.

# Igor Pro Help File

If your XOP is to be used by people other than you, you should create an Igor Pro help file. Igor uses this help file to display help in the Igor Help Browser Command Help tab and to provide templates in Igor procedure windows. You can also ask Igor to display this file when the user clicks the Help button in your dialog or window. The user can open the help file at any time by double-clicking it or using the File->Open File->Help File menu item.

Igor Pro help files work on both Macintosh and Windows. You can edit the file on one platform and use it on both. The help file name should have a ".ihf" extension and on Windows it is required. After editing the file as an Igor formatted notebook, the next time you open it as a help file, Igor will “compile” it.

In Igor Pro 4, you must compile your help file on Macintosh and again on Windows. On Macintosh the help compiler output is stored in the resource fork of the file. On Windows, it is stored in an additional ".ihf.igr" file.

In Igor Pro 5 the help compiler output is stored in the help file itself. You can compile your help file on either platform and it will work on both. No ".ihf.igr" file is needed. Igor Pro 5 can use help files compiled by Igor Pro 4 but not vice versa.

Your help file should include an overview, examples of using your XOP, and a description of each operation and function that your XOP adds to Igor.

It is usually best to start with a help file for one of the sample XOPs and modify it to suit your purposes. For details on creating an Igor Pro help file, see the Igor Pro User’s Manual.

When Igor needs to find an XOP’s help file, to display help in the Help Browser, for example, it looks in the folder containing the executable XOP itself. The sample XOPs use a folder organization, described in Chapter 3, which puts the executable XOP in a development system-specific subfolder. The help file is at the top of the sample XOP folder hierarchy, usually one level up from the executable XOP. With Igor Pro 5.02 or later, you can make Igor find the help file by putting an alias (*Macintosh*) or shortcut (*Windows*) for it in the same folder as the executable XOP. The alias or shortcut must have the exact same name as the help file itself.

# Help for External Operations and Functions

Igor Pro provides help for built-in operations and functions via the Command Help tab of the Igor Help Browser and also via the Templates popup menu in the procedure window. Your XOP can supply this kind of help for its own operations and functions.

When Igor Pro builds the list in the Command Help tab and when it builds the Templates popup menu, it automatically includes any external operations declared in your XOP's XOPC resource and any external functions declared in your XOP's XOPF resource.

When the user chooses an external operation or function that your XOP provides, Igor Pro looks in the folder containing the executable XOP for your XOP's help file. If it finds it, it looks in the help file for a subtopic that matches the operation or function. If it finds this, it displays the help in the Help Browser or displays the template in the procedure window.

Igor Pro 5 looks for an XOP help file, in the same folder as the XOP file itself, with the same name as the XOP file but with " Help.ihf" appended. If the XOP file name is "GBLoadWave.xop", Igor will look for "GBLoadWave Help.ihf". This is what we call the "default help file name". In most case, the default name is fine so this technique will work.

Prior to Igor Pro 5, on Macintosh Igor looked for "GBLoadWave Help" without the extension. If you must run with Igor Pro 4 on Macintosh, omit the extension. This will also work with Igor Pro 5 on Macintosh.

You may want to override the default help file name. For example, you might have an XOP named "GBLoadWave Release" and another XOP named "GBLoadWave Debug", and you want both XOPs to use a single help file named "GBLoadWave Help.ihf". To override the default help file name, you must put a STR# 1101 resource in your XOP's resource fork. Igor takes item number 3 in this resource, if it exists, as the help file name. Here is an example from GBLoadWave.

```
// Macintosh, in GBLoadWave.r.
resource 'STR#' (1101) {   // Misc strings that Igor looks for.
   {
      "-1",
      "---"
      "GBLoadWave Help",   // Name of XOP's help file.
   }
};

// Windows, GBLoadWaveWinCustom.rc.
1101 STR#                  // Misc strings that Igor looks for.
BEGIN
   "-1\0",
   "---\0",
   "GBLoadWave Help\0",    // Name of XOP's help file.

   "\0"                    // 0 required to terminate the resource.
END
```

The first two items in the STR# 1101 resource are not used by modern XOPs. The first item must be -1. It is the third item that defines the custom XOP help file name. Note that the ".ihf" extension is not included in the resource string but is included in the help file name.

If Igor finds the help file using your custom help file name from STR# 1101 or the default help file name, it then looks in the help file for a subtopic whose name is the same as the name of the operation or function for which the user has requested help. If it finds such a subtopic, it displays the subtopic text in the dialog. Note that subtopics must be governed by a ruler whose name is "Subtopic" or starts with the letters "Subtopic". The best way to create your help file is to start with a WaveMetrics help file and modify it.

# Macintosh Balloon Help

Balloon help is a Mac OS 9 technology. As of Carbon and Mac OS X, Apple has dropped balloon help in favor of “help tags”. Because WaveMetrics had spent countless programmer-hours creating balloon help for the hundreds of menu items and thousands of dialog items in Igor, we wrote code that takes balloon help resources for menus and dialogs and displays their contents.

In Igor Pro 4, balloon help resource information is displayed in the Igor Tips windoid. In Igor Pro 5, it is displayed in help tags. For reasons that are not clear to us, menu help tags do not work under Mac OS 9 but do work under Mac OS X. In both versions, the user turns balloon help on by choosing Show Igor Tips from the Help menu.

For technical reasons, extending this WaveMetrics balloon help support to XOPs is not feasible, except in a very limited fashion, as this table illustrates:

| **Item** | **Supported by Igor Pro 4** | **Supported by Igor Pro 5** |
|---|---|---|
| Single menu items<br>(added by XMI1 resource) | Yes | On Mac OS X only |
| XOP main menus<br>(added by XMN1 resource) | Yes | For CFM XOPs only |
| XOP submenus<br>(added by XSM1 resource) | Yes | For CFM XOPs only |
| Dialog items | Yes | No |

Because of the limited and diminishing support for balloon help, we no longer recommend that XOP programmers spend time to create balloon help resources. Advanced programmers may be able to use Apple’s help tags to fill this gap.

This manual documents only balloon help for single Macintosh menu items. If you want to implement balloon help for the main menu or submenu of your CFM XOP, see Chapter 11 in the XOP Toolkit 3.1 manual which is included on the XOP Toolkit CD ROM or contact WaveMetrics support to receive the XOP Toolkit 3.1 manual in PDF form.

If you create your XOP as a Mach-O binary and if you have hmnu or hdlg resources in your .r file, your .r file will not compile. This is because the Carbon framework which supplies header files in Mach-O projects does not include Balloons.r, the balloon help resource definition file. The solution is to remove or ifdef the hmnu or hdlg resources.

# Balloon Help For Macintosh XOP Menu Items

As described in Chapter 8, **Adding Menus and Menu Items**, you can put any number of menu items in built-in Igor menus by using an XMI1 1100 resource. Here is how you can provide balloon help strings.

For each menu item that you add to a built-in Igor Pro menu you can supply a corresponding STR# resource to contain four help strings for that one menu item. The resource ID of the STR# resource is not critical but, by convention, you should start these balloon string resources from ID=1110.

This example comes from the WindowXOP1.r file used in the WindowXOP1 XOP.

```
// Balloon help for the "WindowXOP1" item
resource 'STR#' (1110, "WindowXOP1") {
   {
      /* [1] (used when menu item is enabled) */
      "WindowXOP1 is a sample XOP that adds a simple window to Igor.",

      /* [2] (used when menu item is disabled) */
      "",         /* the item is never disabled */

      /* [3] (used when menu item is checked) */
      "",         /* the item is never checked */

      /* [4] (used when menu item is marked) */
      "",         /* the item is never marked */
   }
};
```

Notice that the resource has a name and that it is identical to the menu item that WindowXOP1 adds to Igor's built-in menu. This is how Igor Pro knows which STR# provides help for which menu item.

If the WindowXOP1 XOP used an XMI1 resource to add a second or third menu item, we would use STR# 1111 and STR# 1112 resources to provide balloon help for these items and we would make sure that the names of these resources matched the the corresponding menu items.

Each STR# resource used to provide balloon help strings must contain exactly four strings. The first string is used when the menu item is enabled. The second is used when it is disabled. The third is used when it is checked and the fourth is used when it is marked. “Marked” means that you have added a symbol other than a checkmark. This is rarely used.

# Status Line Help For Windows XOP Menu Items

You can specify status line help strings for your XOP's menu items and menus. If your XOP adds one or more menu items to built-in Igor Pro menus, you can use a STR# resource to provide status line help for those items. If your XOP uses XMN1 resources to add main menu bar menus or XSM1 resources to add submenus, you can use HMNU and MENUHELP resources to provide balloon help for those menus. Here are the details.

## Status Line Help For Items Added To Igor Menus

For each menu item that you add to a built-in Igor Pro menu you can supply a corresponding STR# resource to contain two help strings for that one menu item. The resource ID for each STR# resource must match the corresponding item number in the XMI1 resource. For example, if your XMI1 resource defines two menu items, then you need two STR# resources, one with ID 1 for the first menu item and the other with ID 2 for the second menu item.

You create the status line help strings by entering text in your XOP's WinCustom.rc file. The following STR# resource from a fictitious XOP illustrates the form of the status line help STR# resource.

```
1 STR#         // Status line help for first item in XMI1 resource.
BEGIN
   // The first string is displayed when the menu item is enabled.
   "Computes the average value of a wave.\0",

   // The second string is displayed when the menu item is disabled.
   "Not available because there are no waves.\0",

   "\0"              // Null required to terminate the resource.
END
```

Igor displays the first string when the user highlights your XOP's menu item and the item is enabled. The second string is displayed when the highlighted item is disabled.

Each string must be terminated with a null character (\0) and there must be a null string ("\0") following the last help string.

If your XOP's menu item can never be disabled, use an empty string ("\0") as the second string.

## Status Line Help For XOP Menus

If your XOP adds one or more main menus or submenus to Igor Pro, you can provide status line help by including one or more HMNU resources in your XOP. Igor uses the information in HMNU resources to find MENUHELP resources containing the status line strings for your menus. You enter the HMNU and MENUHELP resources in your XOP's WinCustom.rc file.

Here is an example of status line help strings for a fictitious XOP that adds a menu containing three menu items.

```
IDR_MENU1 HMNU DISCARDABLE // Status line help for menu.
BEGIN
   4,          // Number of strings in the menu, including the menu title.
   0,          // MENUHELP group. Always zero.

   // Default help string resource number.
   0,0,0L,     // There is no default help string for this menu.

   // Pointers to MENUHELP resources for each item in the menu.
   1,1,0L,     // Help for menu title is in MENUHELP resource 1.
   1,2,0L,     // Help for menu item 1 is in MENUHELP resource 2.
   1,3,0L,     // Help for menu item 2 is in MENUHELP resource 3.
   1,4,0L      // Help for menu item 3 is in MENUHELP resource 4.
END

1 MENUHELP DISCARDABLE  // Status line string for menu title.
BEGIN
   0L,                  // No WinHelp item.
   1L,                  // This resource contains 1 string.
   "The Pele XOP simulates soccer plays using Igor.\0",
END

2 MENUHELP DISCARDABLE  // Status line string for menu item 1.
BEGIN
   0L,                  // No WinHelp item.
   1L,                  // This resource contains 1 string.
   "Creates a new simulation.\0",
END

3 MENUHELP DISCARDABLE  // Status line string for menu item 2.
BEGIN
   0L,                  // No WinHelp item.
   2L,                  // This resource contains 2 strings.
   "Runs the simulation.\0",
   "Not available because no simulation window is active.\0",
END

4 MENUHELP DISCARDABLE  // Status line string for menu item 3.
BEGIN
   0L,                  // No WinHelp item.
   2L,                  // This resource contains 2 strings.
   "Deletes the active simulation.\0",
   "Not available because no simulation window is active.\0",
END
```

The MENUHELP resource contains the actual status line help strings. The HMNU resource is a directory that Igor uses to associate items in a menu with MENUHELP resources. The first line in the HMNU resource is the number of strings in the menu, counting the menu title and each menu item as one string. The second line contains a placeholder which must be zero.

The remaining lines in the HMNU resource point to MENUHELP resources and consist of three numbers. The first number is 1 if there is a MENUHELP resource for the corresponding menu item or 0 if not. The second number is the MENUHELP resource ID or 0 if there is none. The last number is a placeholder and must be 0L.

The next line in the HMNU resource defines a default MENUHELP resource. The default resource, if present, will be used for menu items that have no associated item in the HMNU. The next line in the HMNU specifies the MENUHELP resource for the menu title string. The remaining lines specify the MENUHELP resource for each menu item. You can add items to the menu at runtime, but they will not have status line help.

The first item in a MENUHELP resource is a topic number for context-sensitive help. This is not presently supported and must be 0L.

The second number tells Igor how many strings follow. This should be 2 if the menu item can be disabled and 1 if it can not be disabled. The rest of the resource consists of that number of strings. Note that all strings in the MENUHELP resource have a terminating null character.

## Context-Sensitive Help For Windows XOP Dialogs

Igor dialogs support context-sensitive help accessed via the question-mark icon. The help text is stored in a WinHelp file prepared using the Microsoft Help Workshop. At runtime, Igor calls the WinHelp function when the user clicks the question-mark icon. If your XOP adds a dialog to Igor, you can add context-sensitive dialog help in the same manner. This does not require any coordination with Igor. You implement this kind of help by calling the Windows API WinHelp function in response to a WM_CONTEXTMENU message that the Windows OS sends to your dialog procedure.

## Help For XOP Dialogs and Windows

Balloon help on Macintosh and context-sensitive help on Windows provide help for a specific dialog item or window icon. The user also may need help that explains the purpose of the dialog or window and gives general help for using it. To provide this kind of help, add a Help button to your dialog or window. When the user clicks the button, call the XOPDisplayHelpTopic XOPSupport routine. XOPDisplayHelpTopic displays a topic that you specify from your XOP help file.

*Chapter* 12

# Debugging

Overview ........ 303
Programming Problems ........ 303
  Excessive Use of Global Variables ........ 303
  Uninitialized Variables ........ 304
  Overwriting Arrays ........ 305
  Off By One Errors ........ 306
  Failure To Check Error Codes ........ 306
  Misconceptions About Pointers ........ 307
  Using Memory Blocks That You Have Not Allocated ........ 308
  Using Memory Blocks That You Have Disposed ........ 309
  Disposing Memory Blocks More Than Once ........ 309
  Failure to Dispose Memory Blocks ........ 310
  Disposing Memory Blocks That Don’t Belong to You ........ 311
  Failure to Check Memory Allocations ........ 311
  Dangling Pointer / Heap Scramble Problems ........ 312
    Dereferencing the Handle ........ 313
    Dereferencing the Handle Without Using a Pointer Variable ........ 314
    Locking the Block ........ 315
    Heap Fragmentation ........ 315
    Be Careful About Unlocking a Block ........ 316
    Using MoveLockHandle ........ 317
    Recommendations for Using Handles ........ 317
    Testing for Heap Scramble Problems ........ 317
Debugging Techniques ........ 318
  Recompiling Your XOP ........ 318
  Symbolic Debugging ........ 318
  Debugging Using XOPNotice ........ 319
  Using Macsbug on Mac OS 9 ........ 319

Crash Logs ........ 319
Avoiding Common Pitfalls ........ 320
Check Your Project Setup ........ 320
Get the Message and Parameters from Igor ........ 320
Understand the Difference Between a String in a Handle and a C String .. 321
Choose Distinctive Names ........ 322
Set the Menu ID for MENU Resources ........ 322
Watch Out for Recursion ........ 322
Structure Alignment ........ 322

# Overview

In this chapter we present tips and techniques learned through years of XOP programming that may save you valuable time.

You can reduce the amount of time that you spend debugging by using good programming practices. These include

- Breaking your program up into appropriate modules
- Using clear and descriptive variable and function names
- Always checking for errors returned by functions that you call
- Keeping the use of global variables to a bare minimum
- Carefully proofreading your code immediately after writing it

The best time to find a bug is when you create it. When you write a routine, take a few minutes to carefully proofread it. Be on the lookout for the common errors listed below that can take hours to find later if you don't catch them early.

Most of the problems that people run into in writing XOPs are standard programming problems. We discuss several of them in the next section. The middle part of the chapter discusses debugging techniques. The chapter ends with a discussion of pitfalls that are specific to XOP programming.

# Programming Problems

Here are some of the common programming problems that you should be on the lookout for as you proofread your code.

## Excessive Use of Global Variables

Global variables are bad because any routine in your program can change them and any routine can depend on them. This can lead to complex and unexpected dependencies which in turn leads to bugs. A given routine may change a global variable, having an unforeseen impact on another routine that uses the global. This creates a bug that may manifest itself at unpredictable times.

By contrast, a routine that accesses no global variables depends only on its inputs and can not impact routines other than the one that called it. This makes it easy to verify that the routine works properly and reduces the likelihood of unforeseen effects when you change the routine.

You can avoid using globals by passing all of the necessary information from your higher level routines to you lower level routines using parameters. This does lead to routines with a lot of parameters but this is a small price to pay for robustness.

It is alright to use global variables for things that you can set once at the beginning of your program and then never need to change. For example, the XOPSupport routines use a global variable, XOPRecHandle, to store the handle that Igor uses to communicate with your XOP. This global is set once when your XOP calls XOPInit and then is never changed. Because it is never changed, it can't introduce complex dependencies.

## Uninitialized Variables

The use of uninitialized variables can be difficult to find. Often an uninitialized variable problem shows up only if your code takes a certain execution path. Here is an example.

```
long numBytes;
double* dp;

if (<condition1>)
   numBytes = 100;
else
   if (<condition2>)
      numBytes = 200;

dp = NewPtr(numBytes);      // Possible bug
<Fill block with data>;
```

If neither <condition1> nor <condition2> is true, then numBytes will be uninitialized. This may happen only in rare cases so your code may seem to run fine, but once in a while it crashes or behaves erratically.

Even if numBytes is uninitialized, your code may run fine some times because numBytes just happens to have a value that is sufficient. This makes it even harder to find the problem because it will be very intermittent.

If this bug is symptomatic, the symptom will most likely be a crash. However, the crash may occur some time after this routine executes successfully. The reason is that this routine will clobber the block of memory that falls after the block allocated by NewPtr. You will not actually crash until something tries to use the clobbered block.

To avoid this problem, proofread your code and pay special attention to conditional code, making sure that all variables are initialized regardless of what path execution takes through the code.

## Overwriting Arrays

It is not too difficult to clobber data on the stack or in the heap by overwriting an array. Here is an example.

```
int
Test(char* inputName)
{
   char name[MAX_OBJ_NAME+1];
   Handle aHandle;

   aHandle = NewHandle(100);
   if (aHandle == NULL)
      return NOMEM;
   strcpy(name, inputName);
   strcat(name, "_N");      // Possible bug
   .
   .
   .
}
```

This code will work fine as long as the inputName parameter is less than or equal to MAX_OBJ_NAME-2 characters long. This is likely to be the case most of the time. Once in a while, it may be longer. This will cause the strcat function to overwrite the name array. The effect of this will depend on what follows the name array in the local stack frame.

With most compilers, the aHandle variable will follow the name array on the local stack. Thus, this bug will clobber the aHandle variable. This will most likely cause a crash when the aHandle variable is used or when it is disposed. It could be worse though. It may corrupt the heap when aHandle is used but not cause a crash until later, making it very difficult to track down.

Here is a very insidious case in which the bug may be asymptomatic most of the time. Imagine that the inputName parameter is MAX_OBJ_NAME-1 characters long. Then, the strcat function will use just one more byte than is allocated for the name array. It will write the terminating null character for the name variable in the first byte (the high byte on Macintosh) of the aHandle variable. Since aHandle contains an address in the heap, this high byte will be zero if the address is in the first 16 megabytes of the memory map. In this case, the bug will cause us to write a zero byte on top of a zero byte and it will be asymptomatic. However, if the address is not in the first 16 megabytes of the memory map, the bug will write a zero on top of a non-zero byte and it will be symptomatic.

To avoid this problem, proofread your code and pay special attention to array and string operations, keeping in mind the possibility of overwriting. Read through the code assuming a worst case scenario (e.g., inputName is as long as it possibly could be).

## Off By One Errors

It is very easy and common to do something one time too many or one time too few. For example.

```
int
RotateValuesLeft(float* values, int numValues)
{
   float value0;
   int i;

   value0 = values[0];
   for(i = 0; i < numValues; i++)
      values[i] = values[i+1];      // Bug
   values[numValues] = value0;      // Bug
}
```

We assume that values parameter points to an array of numValues floats. There are two problems here. First, when i reaches its maximum value (numValues-1), values[i+1] accesses a value that is past the end of the array. Second, and more destructive, the last statement clobbers whatever happens to be stored after the values array. If the values array is in the heap, this may cause heap corruption and a crash at some later time. If the values array is on the stack, it may or may not cause a problem, depending on what is stored after the values array and how it is used.

To avoid this problem, proofread your code and pay special attention to what happens the first and last times through a loop. In this example, assume that numValues is some specific number (3, for example) and work through the loop, paying special attention to the last iteration. Also verify that the last element of an array is being set and that no element beyond the last element is being touched. Remember that, for an array of n elements, the first valid index is zero and the last valid index is n-1.

## Failure To Check Error Codes

*Always* check error codes returned from XOPSupport routines (and any other routines, for that matter) and handle errors gracefully. Failure to check error codes can turn a simple problem into a devilish, irreproducible crash. Here is an example.

```
void
BadCode(void)
{
   char* waveData;
   long dims[MAX_DIMENSION_SIZES+1];

   MemClear(dims, sizeof(dims));
   dims[ROWS] = 100;
   dims[COLUMNS] = 100;
   MDMakeWave(&waveH, "wave0", NULL, dims, NT_FP32, 1);
   waveData = WaveData(waveH);
   MemClear(waveData, 100*100*sizeof(float));
}
```

MDMakeWave returns an error code, but this routine ignores it. If memory is low, MDMakeWave may fail, return NOMEM as the error code, and leave waveH undefined. The XOP will crash when it calls WaveData or MemClear. Since this will happen under low memory conditions only, it will happen irreproducibly and the cause will be hard to find.

The code should be written like this.

```
int
GoodCode(void)
{
   char* waveData;
   long dims[MAX_DIMENSION_SIZES+1];
   int err;

   MemClear(dims, sizeof(dims));
   dims[ROWS] = 100;
   dims[COLUMNS] = 100;
   if (err = MDMakeWave(&waveH, "wave0", NULL, dims, NT_FP32, 1))
      return err;
   waveData = WaveData(waveH);
   MemClear(waveData, 100*100*sizeof(float));
   return 0;
}
```

## Misconceptions About Pointers

People who program infrequently in C sometimes forget that a pointer has to point to something. Here is an example of a common error.

```
void
F1(long* lp)
{
   *lp = 0;
}

void
F2_BAD(void)
{
   long* lp1;

   F1(lp1);          // Bug
}
```

The variable lp1 is a pointer to a long so the compiler is happy with this code. At runtime, however, it may cause a crash. The problem is that lp1 is an uninitialized variable. It contains a random value that could point to anything in memory. When F2_BAD calls F1, F1 sets the value pointed to by lp1 to zero. This sets a random 32-bit section of memory to zero. It could be completely asymptomatic or it could cause an immediate crash or it could cause a crash at a later time. It could cause just about anything. It depends on what value happens to be stored in lp1.

This example shows two correct ways to call F1.

```
void
F2_GOOD(void)
{
   long long1;
   long lp1;

   F1(&long1);
   lp1 = (long*)NewPtr(sizeof(long));
   if (lp1 != NULL) {
      F1(lp1);
      DisposePtr((char*)lp1);
   }
}
```

In the first call to F1, we pass the address of the local variable long1 which we have allocated on the local stack. In the second call to F1, we pass the address of a block of memory in the heap that we have allocated using NewPtr.

To avoid problem with pointers, keep in mind that a pointer variable is just a variable that holds the address of some place in memory and that you must set the value of the pointer to make it point to memory that you have allocated before you use it. When you use a pointer, give some thought to whether it points to some space on the local stack (&long1 in F2_GOOD) or to some space in the heap (lp1 in F2_GOOD). This will help you avoid uninitialized pointers.

## Using Memory Blocks That You Have Not Allocated

This is really another case of an uninitialized variable. We make a special case of this because it is a common one. Here is an example.

```
int
Test(int v1, int v2)
{
   Handle h;
   long size;
   int err = 0;

   size = v1 * v2;
   if (size > 0) {
      h = NewHandle(size);
      <Do something with the handle>;
   }

   DisposeHandle(h);       // Possible bug
   return err;
}
```

We dispose the handle even in the case where size <= 0. In that case, we will not have allocated the handle. This may or may not be symptomatic, depending on what value happens to be stored in the handle and on what operating system we are running with. Some operating systems are more tolerant than others of being passed a garbage handle.

This problem is usually a lot more subtle than this example illustrates. The function may have all sorts of conditionals, loops and switches and there may be certain paths of execution in which the handle is not allocated. Often the best solution for this is to use the handle itself as a flag indicating whether or not it has been allocated. For example:

```
int
Test(int v1, int v2)
{
   Handle h;
   long size;
   int err = 0;

   h = NULL;               // Flag that handle has not been allocated
   size = v1 * v2;
   if (size > 0) {
      h = NewHandle(size);
      <Do something with the handle>;
   }

   if (h != NULL)          // Test flag
      DisposeHandle(h);
   return err;
}
```

### Using Memory Blocks That You Have Disposed

### Disposing Memory Blocks More Than Once

These problems are similar to the preceding one in that they generally occur in a complex function that has many potential execution paths. They can also happen if you use global variables. For example:

```
static Handle gH;                // A global handle variable.

void
InitializePart1(void)
{
   long size;

   size = GetHandleSize(gH);     // Uses global before it is initialized.
   memset(**gH, 0, size);
}
```

```
void
InitializePart2(void)
{
   gH = NewHandle(100);          // Initializes global handle.
}

void
main(void)
{
   InitializePart1();
   InitializePart2();
}
```

The problem here is confusion as to which routine is responsible for what. InitializePart1 assumes that gH has already been allocated but it has not. So it is using a garbage handle with unpredictable results but most likely a spectacular crash. This illustrates one of the fundamental problems with global variables - it is difficult to remember when they are valid and what routines have the right to use them.

To avoid this problem, pay special attention to the allocation and deallocation of memory blocks during coding and proofreading. Make sure that each is allocated once and disposed once. In complex situations, initialize pointers and handles to NULL and test for NULL before allocating or disposing the memory to which they refer. After disposing, reset the pointer or handle to NULL.

## Failure to Dispose Memory Blocks

This problem is commonly called a "memory leak". It occurs when you forget to deallocate a block that you have allocated. This is usually not fatal - it just consumes memory unnecessarily.

One way to avoid this problem is to proofread your code carefully, paying special attention to all memory allocations.

Make sure that you know, when you receive a handle from Igor, whether the handle belongs to Igor or is yours to dispose. The XOP Toolkit documentation tells you which of these is the case. For example, when your external function receives a handle containing a string parameter, the handle belongs to you and you must dispose it. However, when your external operation receives a string handle, it belongs to Igor and you must not dispose it. These issues are discussed in Chapter 5 and Chapter 6.

You can check for leaks by writing a loop that calls your XOP over and over while monitoring memory usage. On Mac OS X, you can use Activity Monitor or MallocDebug. On Windows you can use the Task Manager. Third-party leak detection tools are also available.

Keep in mind that some fluctuation in memory allocation is normal. For example, the first time you call your external operation or function, Igor or your XOP may allocate some memory that

needs to be allocated only once. But if memory usage continues to grow without explanation as you call your XOP over and over then you may have a leak.

## Disposing Memory Blocks That Don't Belong to You

If you call GetWave to get a wave handle or call FetchStrHandle to get a string variable handle, you should never dispose that handle. It belongs to Igor and Igor will dispose it. The XOP Toolkit documentation tells you when a handle belongs to Igor and when it is yours to dispose.

A similar situation arises with the Macintosh operating system. Some calls return a handle that belongs to you and is yours to dispose (e.g., NewRgn). Other calls return a handle that belongs to the system (e.g., GetGrayRgn).

## Failure to Check Memory Allocations

If your XOP works fine most of the time but crashes under low memory conditions, it is possible that you have failed to check memory allocations. Here is an example:

```
void
Test()
{
   Handle h;
   h = NewHandle(10000);
   memset(*h, 0, 10000);      // Possible bug
}
```

Under sufficiently low memory, the NewHandle call will fail and h will contain 0. The memset function will attempt to clear the first 10000 bytes starting at address 0 and will cause an access violation exception. The routine should be rewritten like this:

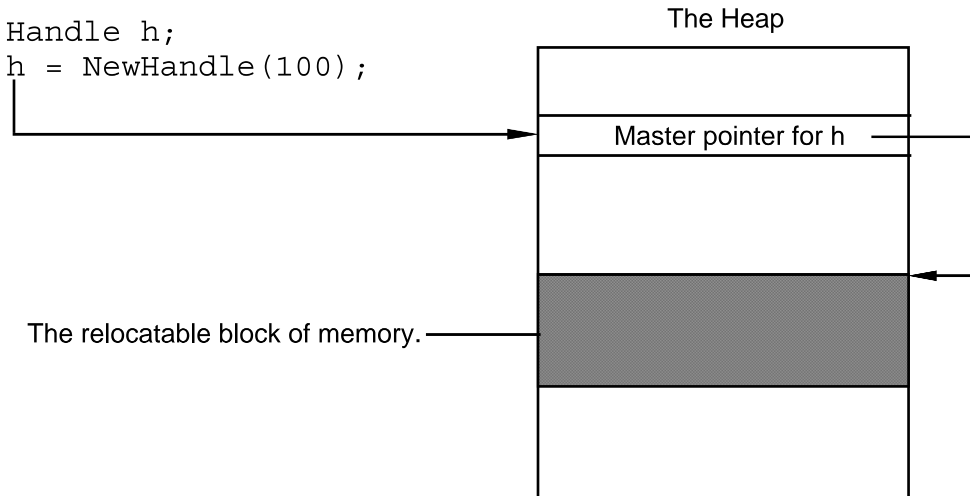
```
int
Test()
{
   Handle h;
   h = NewHandle(10000);
   if (h == NULL)
      return NOMEM;
   memset(*h, 0, 10000);
   return 0;
}
```

Now that Test detects low memory, it will no longer cause an exception. However, the routine that called Test will need to know that an error has occurred and must be able to stop whatever it was doing and return an error also. In fact, all of the routines in the calling chain that led to Test need to be able to cope with an error. For a program to be robust, it must be able to gracefully back out of any operation in the event of an error.

Virtual memory operating systems make it hard to test how your XOP behaves under low memory conditions because memory allocation calls will almost never fail on such systems. In other words, virtual memory may camaflouge bad programming practices. Good code still always checks memory allocations and handles failed allocations gracefully.

## Dangling Pointer / Heap Scramble Problems

A Macintosh handle is a variable that refers to a block of memory in the heap. Unlike a pointer, which points directly to a block of memory, a handle points to the block indirectly. Specifically, it points to a pointer, called a master pointer, to the block of memory.




The benefit of this is that the memory manager can move the block to make room for other blocks. The memory manager updates the master pointer so that it points to the new location of the block. The handle still points to the master pointer and thus is still a valid reference to the block.

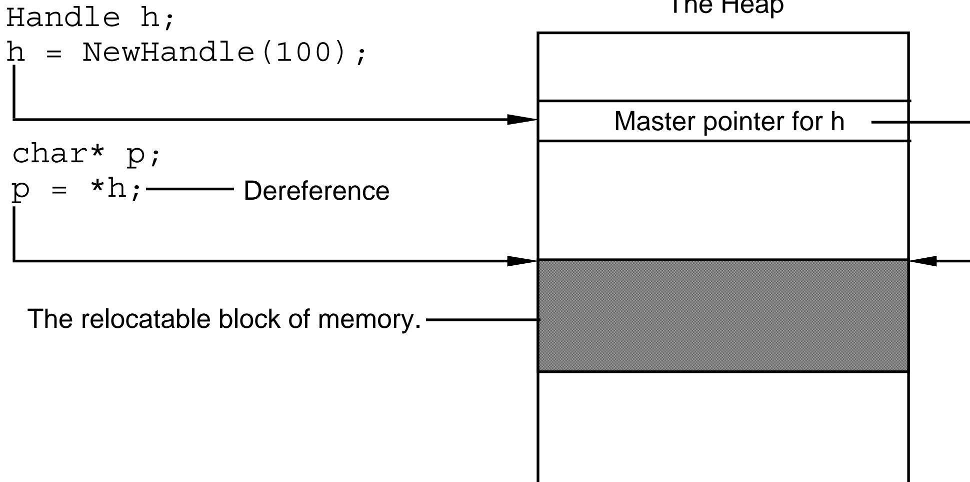
Even Windows XOPs use Macintosh-style handles, as is explained in the section **Data Sharing** on page 139. As of this writing, the Igor emulation of the Macintosh Memory Manager on Windows never relocates blocks of memory. This means that heap scramble problems (described below) can not occur on Windows.

It is hard to tell from Apple’s documentation, but it is likely that the Mac OS X memory manager also does not relocate blocks of memory. That leaves just Mac OS 9 which definitely relocates blocks of memory.

You should write your code as if a block referenced by a handle could be relocated. This will guarantee that your code runs on Mac OS 9 and continue to run if Apple changes the Mac OS X implementation or if WaveMetrics changes the Windows implementation.

### Dereferencing the Handle

To read or write data stored in a block of memory, we need a pointer to it. We obtain the pointer by “dereferencing” the handle, using the C * operator.




The pointer p now points to the block in memory and we can access the data using it.

The problem is that, if the memory manager moves the block in the heap, then p will point to a place in the heap where the data no longer resides. This place may now contain a free block or it may contain the data associated with a different handle. If we use p to read data, we will read garbage. If we use p to write data, we may trash a different handle’s data, causing a problem that will show up later.

Using a pointer obtained by dereferencing a handle after the block of data has been moved by the memory manager is a “dangling pointer” bug or a “heap scramble” bug.

The memory manager will not move the block of memory capriciously. It will only move the block to make room for another block – when the program allocates a new handle or pointer or tries to increase the size of an existing handle or pointer. Thus, you can avoid a dangling pointer bug by not using a pointer to a relocatable block across a call that allocates memory. For example:

```
Handle h1;
char* p1;
Handle h2;

h1 = NewHandle(100);
p1 = *h1;                  // Dereference h1 to point to data.
memset(p1, 0, 100);
h2 = NewHandle(100);       // This allocates memory. p1 is no longer valid.
p1 = *h1;                  // Dereference h1 again to point to data.
```

```
<Use p1 to access data>;
```

We need to dereference h1 the second time because the second call to NewHandle may cause the memory manager to relocate the block of memory that h1 refers to.

In this example, the need to dereference the handle again is obvious. Often, however, it is not obvious at all. For example, you may dereference the handle and then call a subroutine that does not allocate memory and thus does not scramble the heap. Then you use the pointer obtained by dereferencing the handle. Everything works fine. Later, you change the subroutine so that now it *does* allocate memory. You are unlikely to remember that somewhere else in your program, you have made the assumption that the subroutine does not scramble the heap. Now you have a heap scramble bug. Because of this, the technique shown above should be used only with great care.

### Dereferencing the Handle Without Using a Pointer Variable

To avoid having a dangling pointer, it is usually best to dereference the handle and use it in the same statement. This is most applicable when the handle refers to a block that contains a structure. For example, instead of:

```
StructHandle h;         // A handle to a block containing some structure.
StructPtr p;            // A pointer to a block containing the structure.
h = <Get Handle>;
p = *h;                 // Dereference.
p->field1 = 123;
p->field2 = 321;
```

we use:

```
StructHandle h;         // A handle to a block containing some structure.
h = <Get Handle>;
(*h)->field1 = 123;
(*h)->field2 = 321;
```

The benefit of this is that there is no pointer to dangle. However, there is one tricky case in which this will not save us:

```
StructHandle h;         // A handle to a block containing some structure.
h = <Get Handle>;
(*h)->field1 = <Call a routine that scrambles the heap>; // Bug
```

This could cause a dangling pointer/heap scramble bug. It is not obvious from looking at the source code. However, if you looked at the compiled machine code, you might see that the compiler stores *h in a register, calls the subroutine, and finally stores the value returned by the subroutine using the address stored in the register. Because the subroutine caused the heap to scramble, the address in the register no longer points to the block of data associated with the handle.

The solution is to rewrite this as:

```
StructHandle h;          // A handle to a block containing some structure.
int temp;
h = <Get Handle>;
temp =  <Call a routine that scrambles the heap>;
(*h)->field1 = temp;
```

### Locking the Block

To prevent this problem, we can lock the block in memory. For example:

```
Handle h1;
char* p1;

h1 = NewHandle(100);
HLock(h1);               // Lock the block so that it can not move.
p1 = *h1;                // Dereference h1 to point to data.
<Do something>;
HUnlock(h1);
```

### Heap Fragmentation

Locking the block tells the memory manager that it is not allowed to move the block. This solves the heap scramble problem but it introduces another problem. The locked block may be in the middle of the heap. If the <Do something> routine needs to allocate memory, the memory manager will not be able to moved the locked block out of the way to consolidate free space. This is called “heap fragmentation” and it may cause <Do something> to fail.

A solution for this is to move the block away from the middle of the heap before locking it.

```
Handle h1;
char* p1;

h1 = NewHandle(100);
MoveHHi(h1);             // Move the block out of the way.
HLock(h1);               // Lock the block so that it can not move.
p1 = *h1;                // Dereference h1 to point to data.
<Do something>;
HUnlock(h1);
```

MoveHHi moves the block to the top of the heap. This solves the heap fragmentation problem but it introduces another problem. Moving the block to the top of the heap takes time. Thus, it is not a good idea to use MoveHHi in a routine that is frequently-called.

The best solution is to leave the block unlocked and dereference it with care. However, if the code in question is not called from a tight loop, and needs to access many fields in the block of memory, then locking the block is appropriate.

### Be Careful About Unlocking a Block

There is yet another problem that can occur because of locking and unlocking a block. Consider this example:

```
void
F1(Handle h)
{
   char* p;
   HLock(h);
   p = *h;                  // Dereference.
   <Do something with p>;
   HUnlock(h);
}

void
F2(void)
{
   Handle h;
   char* p;

   h = NewHandle(100);
   HLock(h);
   p = *h;                  // Dereference.
   F1(h);
   <Do something with p>;
   HUnlock(h);
}
```

F2 locks and dereferences the handle. It then calls F1 which locks, dereferences, uses and then unlocks the handle. When F1 returns to F2, the handle is unlocked but F2 thinks it is still locked. We have a dangling pointer problem.

One solution for this is to never unconditionally unlock a block. Instead, restore its locked/unlocked state to what it was before you locked it. F1 can be rewritten like this:

```
void
F1(Handle h)
{
   char* p;
   int hState;

   hState = HGetState(h);
   HLock(h);
   p = *h;                  // Dereference.
   <Do something with p>;
   HSetState(h, hState );
}
```

### Using MoveLockHandle

The XOPSupport MoveLockHandle combines the actions of moving the block to the top of the heap, getting its locked/unlocked state, and locking the block. We would use it to write F1 like this:

```
void
F1(Handle h)
{
   char* p;
   int hState;

   hState = MoveLockHandle(h);
   p = *h;                 // Dereference.
   <Do something with p>;
   HSetState(h, hState );
}
```

### Recommendations for Using Handles

Now that we know all of the potential problems in using handles, how can we avoid them? There is no simple answer to this but here are some guidelines.

First, avoid storing a pointer to a relocatable block in a variable. Instead, use the "Dereferencing the Handle Without Using a Pointer Variable" technique described above. If you *must* store a pointer, re-dereference the pointer after calling *any* subroutine that can scramble the heap or could be changed to scramble the heap in the future.

Second, avoid locking handles because this can cause heap fragmentation and also can slow the program down. Instead, use the "Dereferencing the Handle Without Using a Pointer Variable" technique described above. If this is not practical, use the MoveLockHandle/HSetState technique to lock the block while you access it.

### Testing for Heap Scramble Problems

A heap scramble problem can be quite irreproducible and therefore can easily escape detection. The reason for this is that routines that *can* scramble the heap often do not scramble the heap. For example, a call to NewHandle can scramble the heap but if the memory manager can allocate the requested space without scrambling the heap, it will. Thus, it is very easy to have heap scramble bugs in your program that manifest themselves sporadically.

On Mac OS 9, the Macsbug debugger has a heap scramble feature. You turn heap scramble on by executing the HS command from Macsbug. Heap scramble causes the heap to scramble any time your program calls a routine that *possibly could* scramble the heap. Thus, if you have a potential heap scramble problem, it is likely to become symptomatic when you run with heap scramble on.

There is no such tool for Mac OS X.

# Debugging Techniques

This section discusses several methods for debugging an XOP:

- Using a symbolic debugger
- Using XOPNotice (equivalent to a Print statement)
- Using a low-level debugger such as Macsbug
- Using a crash log

This section does not cover the most effective debugging method: carefully reading your code. Before you spend time debugging, a simple careful reading of your code is the most effective way to find bugs. If that fails, the techniques discussed here are useful for narrowing down the suspect area of your code.

Debugging a problem that you can reproduce at will is usually not too difficult.

If you have an intermittent problem, it is useful to look for a simple set of steps that you can take to make the problem happen. If you can make the problem happen, even one time in ten, with a simple set of steps, this allows you to rule out a lot of potential causes.

Sometimes bugs are so intermittent that there is little hope of catching them red-handed. For bugs that cause a crash, a crash log, described on page 319, may help you to identify the problem area.

## Recompiling Your XOP

During the debugging process, you often need to recompile your XOP and try it again. You must quit Igor to recompile your XOP.

It is generally best to leave your executable XOP file in your project folder during development. Make an alias (*Macintosh*) or shortcut (*Windows*) for your XOP file and drag the alias or shortcut into the Igor Extensions folder to activate your XOP. This saves you the trouble of dragging the XOP file into the Igor Extensions folder each time you recompile the XOP.

## Symbolic Debugging

Usually, the debugging method of choice is to use the symbolic debugger that comes with your development system. Chapter 3 includes a discussion on using the symbolic debuggers of various development systems.

Sometimes symbolic debugging does not work because the problem does not manifest itself when running the debugger or because the bug crashes the debugger. For these cases, the following debugging techniques are available.

## Debugging Using XOPNotice

The simplest tool for debugging an XOP is the XOPNotice XOPSupport routine. This lets you print a message in Igor's history area so that you can examine intermediate results and see how far your XOP is getting. For example:

```
char temp[256];
sprintf(temp, "Wave type = %d"CR_STR, WaveType(wavH));
XOPNotice(temp);
```

**NOTE**: When Igor passes a message to you, you *must* get the message, using GetXOPMessage, and get all of the arguments, using GetXOPItem, *before* doing any callbacks, including XOPNotice. The reason for this is that the act of doing the callback overwrites the message and arguments that Igor is passing to you.

## Using Macsbug on Mac OS 9

Macsbug is a free low-level debugger made by Apple Computer. It runs on Mac OS 9 only, not on Mac OS X. The nice thing about Macsbug is that it is very unintrusive. It does not require a lot of memory and does not affect the environment in which your XOP executes. It just sits there quietly until you invoke it or you crash.

Some of the most useful Macsbug commands are:

| Command | Meaning | What It Does |
|---|---|---|
| HC | Heap check | Tells you if the heap is corrupted. |
| SC | Stack crawl | Shows the chain of functions leading to the one that crashed. |
| STDLOG | Standard Log | Generates a log containing information on the state of the machine. |
| HS | Heap Scramble | Causes heap scramble problems to become symptomatic by scrambling the heap any time a memory allocation is made. |

See Chapter 12 in the XOP Toolkit 3.1 manual for further on Macsbug. The XOP Toolkit 3.1 manual is included on the XOP Toolkit CD ROM or contact WaveMetrics support to receive it in PDF form.

## Crash Logs

On Mac OS X, Windows 2000 and Windows XP, the operating system records information about the state of the machine when a crash occurs. This happens only if a debugger, such as the CodeWarrior or Visual C++ debugger, has not handled the crash. Thus the crash log may be

useful to help a programmer (such as you) understand what went wrong when a crash occurs on the machine of a non-programmer (such as the people who use your XOP).

The most useful part of the log is the “stack crawl” or “stack back trace” which shows the calling chain of functions leading to the instruction that crashed. Unfortunately the information in the stack crawl is not always meaningful. Sometimes it is garbage but sometimes it provides a useful clue. Sometimes it is not garbage but is too complicated to be of much use.

On Mac OS X, the crash log is in a file named “Igor Pro.crash.log” (assuming that the Igor application files is named “Igor Pro”). You can view this file by running Apple’s Console program and clicking the Logs icon or by searching for the file in the Finder and opening it as a text file.

On Windows 2000 and XP, the crash log file is named “drwtsn32.log”. It is a plain text file that you can view in a text editor. The best way to find it is to do a file search from the desktop. On Windows XP, the crash log file is hidden and will not be found unless you use the advanced search settings to include hidden files and folders in the search. For some unknown reason, the stack crawl (called “stack back trace”) in a Dr. Watson log file is usually meaningless. See “Dr. Watson” in the Windows help for additional information.

# Avoiding Common Pitfalls

This section lists some common problems that are specific to XOP programming.

## Check Your Project Setup

See Chapter 3 for instructions on setting up CodeWarrior, Xcode and Visual C++ projects.

When in doubt, compare each of the settings in your project to the settings of an XOP Toolkit sample project.

If you are getting link errors, check the libraries in your project and compare them to the sample projects.

## Get the Message and Parameters from Igor

When Igor passes a message to you, you *must* get the message, using GetXOPMessage, and get all of the arguments, using GetXOPItem, *before* doing any callbacks. The reason for this is that the act of doing the callback overwrites the message and arguments that Igor is passing to you.

This problem often occurs when you put an XOPNotice call in your XOP for debugging purposes. XOPNotice is a callback and therefore clobbers the arguments that Igor is sending to

your XOP. Don't call XOPNotice or any other callback until you have gotten the message and all parameters from Igor.

## Understand the Difference Between a String in a Handle and a C String

When dealing with strings of indeterminate length, Igor uses a handle to contain the string's characters. Igor passes a string parameter to external operations and functions in a handle. There are other times when you receive a string in a handle from Igor. A string in a handle contains no null terminator. To find the number of characters in the string, use GetHandleSize.

A C string requires a null terminator. Thus, you can not use a string in a handle as a C string. Instead, you must copy the contents of the handle to a buffer and then null-terminate the buffer. The XOPSupport functions GetCStringFromHandle and PutCStringInHandle may be of use in cases where you can set an upper limit to the length of the string. Make sure that your buffer is large enough for the characters in the handle plus the null terminator.

Another approach is to append a null character to the handle, lock it, use it as a C string, remove the null character, and unlock it. This is useful for cases where you can not set an upper limit on the length of the string. For example:

```
Handle h;
long len;
char* p;
int hState;

h = <A routine that returns a string in a handle>;
if (h != NULL) {
   len = GetHandleSize(h);
   SetHandleSize(h, len+1);      // Make room for null terminator.
   if (MemError())
      return NOMEM;
   *(*h+len) = 0;                // Null terminate the text.
   hState = MoveLockHandle(h);
   p = *h;
   <Use p as a C string>;
   HSetState(h, hState);
   SetHandleSize(h, len);        // Remove null terminator.
}
```

There are some times when you are required to pass a string in a handle back to Igor. Again, the handle must contain just the characters in the string, without a null terminator. On the other hand, if you are passing a C string to Igor, the string must be null-terminated.

## Choose Distinctive Names

If your XOP adds an operation or function to Igor you should be careful to choose a name that is unlikely to conflict with any built-in Igor names or user-defined names. See **Choose a Distinctive Operation Name** on page 150 and **Choose a Distinctive Function Name** on page 186 for details.

## Set the Menu ID for MENU Resources

If you have a MENU resource in your Macintosh XOP, make sure to set the menu's menu ID field. See **Menu IDs Versus Resource IDs** on page 238 for details.

## Watch Out for Recursion

If your XOP adds a window to Igor, the DoUpdate, XOPCommand, XOPSilentCommand, and SpinProcess XOPSupport routines can cause Igor to call your XOP while your XOP is already running. See **Handling Recursion** on page 137 for details.

## Structure Alignment

Make sure that all structures that might be shared with Igor are 2-byte-aligned. See **Structure Alignment** on page 279 for details.

*Chapter* 13

# XOPSupport Routines

About XOPSupport Routines ... 325
Routines for Communicating with Igor ... 326
Operation Handler Routines ... 329
Routines for Parsing Commands ... 335
Routines for Accessing Waves ... 336
Routines for Accessing Variables ... 368
Routines for Accessing Data Folders ... 376
Routines for XOPs with Menu Items ... 392
Routines for XOPs that Have Dialogs ... 398
  Dialog Popup Menus ... 405
Routines for XOPs that Access Files ... 411
Routines for File-Loader XOPs ... 426
Routines for XOPs with Windows ... 430
Routines for XOPs with Text Windows ... 435
  Creating and Disposing Text Windows ... 435
  Responding to Text Window Messages ... 436
  Text Window Utility Routines ... 442
Routines for Dealing with Resources ... 447
Routines for XOPs That Use FIFOs ... 448
Numeric Conversion Routines ... 450
Routines for Dealing With Object Names ... 455
Color Table Routines ... 461
Routines for Dealing With Igor Procedures ... 464
Windows-Specific Routines ... 473
Miscellaneous Routines ... 476
Programming Utilities ... 491
Macintosh Emulation Routines ... 497
  Emulated Macintosh Memory Management Routines ... 497

Emulated Menu Management Routines ..................................................... 502
Miscellaneous Emulated Macintosh Routines ........................................... 504

# About XOPSupport Routines

The files in the XOPSupport folder define a large number of routines that communicate with Igor or provide a utility. Most XOPs will use only a small fraction of these routines. These routines are described in detail in this chapter. The chapter presents the routines in functional groups.

Some routines work only with certain versions of Igor Pro. XOP Toolkit 5 supports Igor Pro 4 or later so issues involving earlier versions of Igor are not discussed below. Routines that require Igor Pro 5 are so noted. See **Igor/XOP Compatibility Issues** on page 140 for further information on compatibility.

# Routines for Communicating with Igor

These routines provide communication between your XOP and Igor.

```
void
XOPInit(ioRecHandle)
IORecHandle ioRecHandle;   // Used to pass info between Igor and XOP
```

The XOP must call XOPInit from its main routine, before calling any other callbacks.

XOPInit stores the ioRecHandle in the global variable XOPRecHandle. All XOPSupport routines that communicate with Igor use this global variable. You do not need to access it directly.

```
void
SetXOPType(type)
long type;                 // Defines capabilities and mode of XOP
```

type is some combination of TRANSIENT, RESIDENT, IDLES, and ADDS_TARGET_WINDOWS which are bitwise flags defined in XOP.h.

By default, your XOP is resident, meaning that Igor will leave it in memory once it is invoked. Except in very rare cases, you should leave your XOP as resident.

You can call SetXOPType(TRANSIENT) when your XOP is finished and you want Igor to close it. At one time this was recommended as a memory saving device. Now, however, memory is more plentiful and making your XOP transient is likely to cause more trouble than it prevents. XOPs that define external operations or functions must not be transient because they need to remain memory-resident. Igor Pro 5 ignores this call if your XOP adds an external operation or function.

Call SetXOPType(RESIDENT | IDLES) if you want to get periodic IDLE messages from Igor. This is appropriate, for example, in data acquisition XOPs.

The ADDS_TARGET_WINDOWS bit has to do with XOP target windows. Most XOPs will not need to set this bit. See **Adding XOP Target Windows** on page 257 for further information.

See **The IORecHandle** on page 135 for details.

```
void
SetXOPEntry(entryPoint)
ProcPtr entryPoint;        // Routine to handle all messages after INIT
```

Identifies to Igor the routine in your XOP which services all messages after INIT.

You must call this from your main function, passing it the address of your XOPEntry routine.

```
void
SetXOPResult(result)
long result;                // Result of XOP operation
```

Sets the result field of your XOP's IORecHandle.

Igor looks at the result field when your XOP returns. The result is usually an error code in response to the CMD, FUNCTION or MENUITEM messages from Igor. For other messages from Igor, the result may be some other value.

Do not use SetXOPResult to return errors from a direct external operation or function. For a direct operation or function, the CMD or FUNCTION message is not sent to your XOPEntry routine. "Direct" means that your function is called directly from Igor, not through your XOPEntry routine. Such functions return result codes as the function result.

As of Igor Pro 5, most new external operations and functions will be direct.

See **XOP Errors** on page 127 for details on error codes.

```
long
GetXOPResult(void)
```

Returns the contents of the result field of your XOP's IORecHandle.

This is used by XOPSupport routines to get the result of a callback to Igor. You should not call it yourself.

```
void
SetXOPMessage(message)
int message;                // Message to pass to Igor during callback
```

Sets the message field of your XOP's IORecHandle.

This is used by XOPSupport routines during callbacks to Igor. You should not call it yourself.

```
long
GetXOPMessage(void)
```

Returns the message field of your XOP's IORecHandle.

The message field contains the message that Igor is sending to your XOP. Your XOPEntry routine must call GetXOPMessage to find out what message Igor is sending.

**NOTE**: The message field is reused each time Igor calls your XOPEntry routine with a message or you call Igor back. Therefore, you must call GetXOPMessage before you do any callbacks to Igor. An easy mistake to make is to do an XOPNotice callback for debugging before you get the message. See **Messages, Arguments and Results** on page 136 for details.

```
void
SetXOPRefCon(refCon)
long refCon;                // Something XOP wants to store
```

Sets the refCon field of your XOP's IORecHandle.

Your XOP can store anything it wants in this field. This call is rarely used because there is no particular reason to use it rather than a global variable.

```
long
GetXOPRefCon(void)
```

Returns the refCon field of your XOP's IORecHandle.

```
long
GetXOPStatus(void)
```

Returns the status field of your XOP's IORecHandle.

This is infrequently-used. It can tell you if Igor is in the background (not the active application).

See the discussion of status field under **The IORecHandle** on page **135**.

```
long
GetXOPItem(itemNumber)
int itemNumber;
```

Returns an item from the item list of XOP's IORec.

itemNumber is zero-based.

If itemNumber is greater than the number of items passed by Igor less one, GetXOPItem returns zero.

This is used by an XOP to get arguments associated with a message from Igor.

**NOTE**: The item list is reused each time Igor calls your XOPEntry routine with a message or you call Igor back. Therefore, if you need to get an item in response to a message from Igor, get it *before* you do *any* callbacks to Igor. An easy mistake to make is to do an XOPNotice callback for debugging before you get all of the arguments. See **Messages, Arguments and Results** on page 136 for details.

# Operation Handler Routines

As of Igor Pro 5 and XOP Toolkit 5, XOPs implement external operations using Igor's Operation Handler, as described in Chapter 5. This section documents XOPSupport routines used in conjunction with Operation Handler.

```
int
RegisterOperation(cmdTemplate, runtimeNumVarList, runtimeStrVarList,
                     runtimeParamStructSize, runtimeAddress, options)
const char* cmdTemplate;            // Specifies operation's syntax
const char* runtimeNumVarList;      // List of numeric output variables
const char* runtimeStrVarList;      // List of string output variables
int runtimeParamStructSize;         // Size of runtime parameter structure
void* runtimeAddress;               // Address of operation Execute
routine
int options;                        // Flags
```

Registers an XOP operation with Operation Handler.

cmdTemplate specifies the operation name and syntax.

runtimeNumVarList is a semicolon-separated list of numeric variables that the operation sets at runtime or NULL if it sets no numeric variables.

runtimeStrVarList is a semicolon-separated list of string variables that the operation sets at runtime or NULL if it sets no string variables.

runtimeParamStructSize is the size of the runtime parameter structure for the operation.

runtimeAddress is the address of the ExecuteOperation function for this operation.

options is reserved for future use. Pass zero for this parameter.

Returns 0 or an error code.

Added for Igor Pro 5.0. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

***Example***

```
char* cmdTemplate;            // From SimpleLoadWaveOperation.c
char* runtimeNumVarList;
char* runtimeStrVarList;

cmdTemplate = "SimpleLoadWave /A[=name:baseName] /D /I /N[=name:baseName] /O
                              /P=name:pathName /Q /W [string:fileParamStr]";
runtimeNumVarList = "V_flag;";
runtimeStrVarList = "S_path;S_fileName;S_waveNames;";
return RegisterOperation(cmdTemplate, runtimeNumVarList, runtimeStrVarList,
            sizeof(SimpleLoadWaveRuntimeParams), ExecuteSimpleLoadWave, 0);
```

```
int
SetOperationNumVar(varName, dval)
const char* varName; // C string containing name of variable
double dval;         // Value to store in variable
```

SetOperationNumVar is used only to implement an operation using Operation Handler. It is used to store a value in an external operation output numeric variable such as V_flag.

varName must be the name of a numeric variable that you specified via the runtimeNumVarList parameter to RegisterOperation.

dval is the value to be stored in the named variable.

If your operation was invoked from the command line, SetOperationNumVar sets a global variable in the current data folder, creating it if necessary.

If your operation was invoked from a macro, SetOperationNumVar sets a macro local variable.

If your operation was invoked from a user function, SetOperationNumVar sets a function local variable.

Returns 0 or an error code.

Added for Igor Pro 5.0. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

***Example***

```
SetOperationNumVar("V_VDT", number);      // From VDTOperations.c
```

```
int
SetOperationStrVar(varName, str)
const char* varName; // C string containing name of variable
const char* str;     // Value to store in variable
```

SetOperationStrVar is used only to implement an operation using Operation Handler. It is used to store a value in an external operation output string variable such as S_fileName.

varName must be the name of a string variable that you specified via the runtimeStrVarList parameter to RegisterOperation.

str points to the value to be stored in the named variable.

If your operation was invoked from the command line, SetOperationStrVar sets a global variable in the current data folder, creating it if necessary.

If your operation was invoked from a macro, SetOperationStrVar sets a macro local variable.

If your operation was invoked from a user function, SetOperationStrVar sets a function local variable.

Returns 0 or an error code.

Added for Igor Pro 5.0. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

***Example***

```
SetOperationStrVar("S_VDT", list);        // From VDTOperations.c
```

```
int
VarNameToDataType(varName, dataTypePtr)
const char* varName; // C string containing name of variable or binary
data
int* dataTypePtr;    // Output: Data type of variable
```

This is used only to implement an operation using Operation Handler. It must be called only from your ExecuteOperation function.

This function is used only for operations which take the name of a variable as a parameter and then set the value of that variable. An example is Igor's Open operation which takes a "refNum" parameter.

After calling VarNameToDataType you would then call StoreNumericDataUsingVarName or StoreStringDataUsingVarName.

varName must be a pointer to a varName field in your operation's runtime parameter structure when a pointer to this structure has been passed by Igor to your ExecuteOperation function. Igor relies on the varName field being set correctly (i.e., as Igor set it before calling your ExecuteOperation function). If your operation was called from a user function, the varName field will actually contain binary data used by Igor to access function local variables, NVARs and SVARs.

On output, *dataTypePtr will contain one of the following:

| | |
|---|---|
| 0 | Means varName refers to a string variable or SVAR. |
| NT_FP64 | Means varName refers to a scalar local variable or NVAR. |
| NT_FP64 \| NT_CMPLX | Means varName refers to a complex local variable or NVAR. |

Returns 0 or an error code.

Added for Igor Pro 5.0. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

***Example***

```
See VDTReadBinary.c.
```

```
int
StoreNumericDataUsingVarName(varName, realPart, imagPart)
const char* varName; // C string containing name of variable or binary
data
double realPart;
double imagPart;
```

Stores data in a numeric variable which may be local or global.

This is used only to implement an operation using Operation Handler. It must be called only from your ExecuteOperation function.

This function is used only for operations which take the name of a numeric variable as a parameter and then set the value of that variable. An example is Igor's Open operation which takes a "refNum" parameter.

varName must be a pointer to a varName field in your operation's runtime parameter structure when a pointer to this structure has been passed by Igor to your ExecuteOperation function. Igor relies on the varName field being set correctly (i.e., as Igor set it before calling your ExecuteOperation function). If your operation was called from a user function, the varName field will actually contain binary data used by Igor to access function local variables, NVARs and SVARs.

You should call this routine only after you have determined that varName refers to a numeric variable or NVAR. This will be the case if VarNameToDataType returns a non-zero number as the data type.

Returns 0 or an error code.

Added for Igor Pro 5.0. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

***Example***

```
See VDTReadBinary.c.
```

```
int
StoreStringDataUsingVarName(varName, buf, len)
const char* varName; // C string containing name of variable or binary
data
const char* buf;     // String contents
long len;            // Length of string
```

Stores data in a string variable which may be local or global.

This is used only to implement an operation using Operation Handler. It must be called only from your ExecuteOperation function.

This function is used only for operations which take the name of a string variable as a parameter and then set the value of that variable. An example is Igor's FReadLine operation which takes a "stringVarName" parameter.

varName must be a pointer to a varName field in your operation's runtime parameter structure when a pointer to this structure has been passed by Igor to your ExecuteOperation function. Igor relies on the varName field being set correctly (i.e., as Igor set it before calling your ExecuteOperation function). If your operation was called from a user function, the varName field will actually contain binary data used by Igor to access function local variables, NVARs and SVARs.

You should call this routine only after you have determined that varName refers to a string variable or SVAR. This will be the case if VarNameToDataType returns a zero as the data type.

Returns 0 or an error code.

Added for Igor Pro 5.0. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

***Example***

```
See VDTReadBinary.c.
```

```
int
SetOperationWaveRef(waveH, waveRefIndentifier)
waveHndl waveH;            // The destination wave you created.
int waveRefIndentifier;    // Tells Igor where local wave ref is stored.
```

Sets a wave reference in an Igor user-defined function to refer to a destination wave that your operation created.

This is used only to implement an operation using Igor's Operation Handler. It must be called only from your ExecuteOperation function.

This function is used only for operations which use a DataFolderAndName parameter (described on page 166) and declare it as a destination wave parameter using this kind of syntax in the operation template:

```
SampleOp DataFolderAndName:{dest,<real> or <complex> or <text>}
```

When you use this syntax and your operation is compiled in a user-defined function, the Igor function compiler may automatically create a wave reference in the function for the destination wave. However the automatically created wave reference will be NULL until you set it by calling SetOperationWaveRef.

The SetOperationWaveRef callback sets the automatically created wave reference to refer to a specific wave, namely the wave that you created in response to the DataFolderAndName parameter. You must call it after successfully creating the destination wave.

Igor will create an automatic wave reference only if the operation is called from a user-defined function and only if the destination wave is specified using a simple name (e.g., wave0 but not root:wave0 or $destWave). You have no way to know whether an automatic wave reference was created so you must call SetOperationWaveRef in all cases. SetOperationWaveRef will do nothing if no automatic wave reference exists.

The waveRefIndentifier parameter allows Igor to determine where in memory the wave reference is stored. This information is passed to your XOP in the "ParamSet" field of your ExecuteOperation structure.

Your code should look something like this:

```
// In your RuntimeParams structure
DataFolderAndName dest;
int destParamsSet[1];

// In your ExecuteOperation function
destWaveH = NULL;
err = MDMakeWave(&destWaveH, p->dest.name, p->dest.dfH, dimensionSizes, type,
overwrite);
if (destWaveH != NULL) {
   int waveRefIndentifier = p->destParamsSet[0];
   err = SetOperationWaveRef(destWaveH, waveRefIndentifier);
}
```

Returns 0 or an error code.

Added for Igor Pro 5.04B05. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

# Routines for Parsing Commands

As of Igor Pro 5 and XOP Toolkit 5, XOPs implement external operations using Igor's Operation Handler, as described in Chapter 5. Operation Handler automatically takes care of parsing the command and converting its parameters into useable C variables.

Before Operation Handler, the XOP programmer was required to parse the command. A large number of XOPSupport routines were available to carry out this often complex task. Those routines are still supported by the XOP Toolkit and are still needed by XOPs that have not been updated to use Operation Handler.

To reduce clutter, documentation for these old parsing routines is omitted from this manual. If you need this documentation, please see Chapter 13 in the XOP Toolkit 3.1 manual. The XOP Toolkit 3.1 manual is included on the XOP Toolkit CD ROM or contact WaveMetrics support to receive it in PDF form.

Here is a list of the old parsing routines:

| | |
|---|---|
| **Capitalize** | |
| **GetSymb** | **NextSymb** |
| **IsStringExpression** | |
| **GetFlag** | **GetTrueOrFalseFlag** |
| **FileLoaderGetOperationFlags** | **FileLoaderGetOperationFlags2** |
| **GetName** | **GetDataFolderAndName** |
| **GetDataFolder** | |
| **Keyword** | **GetKeyword** |
| **GetWave** | **GetWaveName** |
| **GetWaveList** | |
| **GetWaveRange** | **CalcWaveRange** |
| **GetNum** | **GetNum2** |
| **GetFlagNum** | **GetLong** |
| **GetAString** | **GetAStringInHandle** |
| **GetPath** | **GetFormat** |
| **GetNumVarName** | **GetStrVarName** |
| **CheckTerm** | **AtEndOfCommand** |

# Routines for Accessing Waves

These routines allow you to create, change, kill or access information about waves. Some require that you pass the wave's name. Others require that you pass a handle to the wave's data structure. You get this handle using the GetWave, GetWaveList, FetchWave, MakeWave, MDMakeWave, GetDataFolderObject or FetchWaveFromDataFolder callbacks.

So that your XOP will work with future versions of Igor, it must not access the contents of wave handles directly but instead must use the routines in this section. Also, wave handles belong to Igor and your XOP must not delete or directly modify them.

Igor Pro 3.0 added multi-dimensional waves, up to four dimensions, as well as waves containing text rather than numbers. To support this, we added a number of XOPSupport routines to the XOP Toolkit. The added routines have names that start with "MD" (e.g., MDMakeWave). These routines have several purposes:

- To get and set multi-dimensional wave properties such as units, scaling and dimension labels.
- To get and set multi-dimensional wave data values, text as well as numeric.
- To provide faster access to wave information than previous XOPSupport routines provided.

All of the old wave access routines will continue to work with all supported versions of Igor. However, for new programming, we recommend that you use the new wave access routines.

The symbols ROWS, COLUMNS, LAYERS and CHUNKS are defined in IgorXOP.h as 0, 1, 2 and 3 respectively. These symbols are often used to index into an array of dimension sizes or wave element indices.

All of the wave access routines are defined in the file XOPWaveAccess.c.

The WaveAccess sample XOP contains examples that illustrate most of the wave access techniques.

In the following list of wave access routines, the older routines are listed first, followed by the newer, multi-dimensional-aware routines.

```
int
MakeWave(waveHandlePtr, waveName, numPoints, type, overwrite)
waveHndl* waveHandlePtr;    // Place to put waveHndl for new wave
char* waveName;             // C string containing name of Igor wave
long numPoints;             // Desired length of wave
int type;                   // Numeric type (e.g., NT_FP32, NT_FP64)
int overwrite;              // If non-zero, overwrites existing wave
```

Creates a 1D wave with the specified name, number of points, and numeric type.

If successful, it returns 0 as the function result and puts a waveHndl for the new wave in *waveHandlePtr.

If unsuccessful, it returns an Igor error code as the function result.

If the numPoints parameter is zero, the wave is created with the default number of points (usually 128) and X scaling (usually x0=0, dx=1). This behavior was defined at a time when Igor required that a wave contain at least two points (prior to Igor Pro 3.0). Because of it, you can not use MakeWave to make a wave with zero points. You must use MDMakeWave for that.

type is one of the following:

| Type | What It Means |
|---|---|
| NT_FP32 | 32 bit floating-point |
| NT_FP64 | 64 bit floating-point |
| NT_I8 | 8 bit signed integer |
| NT_I16 | 16 bit signed integer |
| NT_I32 | 32 bit signed integer |
| NT_I8 \| NT_UNSIGNED | 8 bit unsigned integer |
| NT_I16 \| NT_UNSIGNED | 16 bit unsigned integer |
| NT_I32 \| NT_UNSIGNED | 32 bit unsigned integer |
| TEXT_WAVE_TYPE | Text (string data) |

To make a complex wave, OR with NT_CMPLX, for example (NT_FP64 | NT_CMPLX). Any numeric wave can be complex. Text waves can not be complex so don’t use (TEXT_WAVE_TYPE | NT_CMPLX).

As of this writing, Igor can not overwrite a text wave with a numeric wave or vice-versa and you will receive an error if you attempt to do this.

It is recommended that you use the integer types only to store raw data that you have acquired and when economical storage is a high priority, such as when storing images or other large data sets. The reason for this is that Igor needs to translate integer wave data into floating point for display, analysis or just about any other purpose.

```
int
ChangeWave(waveHandle, numPoints, type)
waveHndl waveHandle;        // WaveHndl for wave to change
long numPoints;             // Desired new length of wave
int type;                   // Data type (e.g., NT_FP32, NT_FP64)
```

Changes the 1D wave to the desired length and data type.

Igor Pro can not change a text wave into a numeric wave or vice-versa and you will receive an error if you attempt to do this.

The function result is 0 if successful or an Igor error code otherwise.

```
int
KillWave(waveHandle)
waveHndl waveHandle;        // waveHndl for wave to kill
```

Kills the specified wave.

The function result is 0 if successful or an Igor error code otherwise.

A wave is in use and can't be killed if it is used in a graph, table or user-defined function. An XOP can also prevent a wave from being killed by responding to the OBJINUSE message from Igor.

```
waveHndl
FetchWave(waveName)
char* waveName;             // C string containing name of Igor wave
```

Returns a handle to the data structure for the named wave in the current data folder or NULL if the wave does not exist.

Be sure to check the wave's type, using WaveType, and return an error if you receive a wave as a parameter whose type you can't handle. You may also want to check the dimensionality of the wave using MDGetWaveDimensions.

To support data folders and liberal wave names, modern XOPs must use FetchWaveFromDataFolder instead of FetchWave.

```
waveHndl
FetchWaveFromDataFolder(dataFolderH, waveName)
DataFolderHandle dataFolderH;
char* waveName;
```

FetchWaveFromDataFolder returns a handle to the specified wave in the specified data folder or NULL if the wave does not exist.

If dataFolderH is NULL, it uses the current data folder.

Be sure to check the wave's type, using WaveType, and return an error if you receive a wave as a parameter whose type you can't handle. You may also want to check the dimensionality of the wave using MDGetWaveDimensions.

```
int
WaveType(waveHandle)
waveHndl waveHandle;        // Handle to wave's data structure
```

Returns the wave's data type.

See MakeWave for a list of data types.

**NOTE**: Future versions of Igor may support other data types. You should make sure that you can handle the type of wave you receive as a parameter in an operation or function and generate an error if you receive a wave type you can't handle.

```
long
WavePoints(waveHandle)
waveHndl waveHandle;        // Handle to wave's data structure
```

Returns the number of points in the wave.

If the wave is multi-dimensional, this is the number of elements in all dimensions. To find the number of elements in each dimension separately, use MDGetWaveDimensions.

```
unsigned long
WaveModDate(waveHandle)
waveHndl waveHandle;        // Handle to wave's data structure
```

Returns wave modification date. This is an unsigned long in Igor date/time format, namely the number of seconds since midnight, January 1, 1904.

The mod date has a resolution of one second.

Modification date tracking was added to Igor in Igor 1.2. If a wave is loaded from a file created by an older version of Igor, the mod date field will be zero and this routine will return zero.

```
int
WaveModCount(waveHandle)
waveHndl waveHandle;       // Handle to wave's data structure
```

Returns a value that can be used to tell if a wave has been changed between one call to WaveModCount and another.

The exact value returned by WaveModCount has no significance. The only valid use for it is to compare the values returned by two calls to WaveModCount. If they are the different, the wave was changed in the interim.

***Example***

```
waveModCount1 = WaveModCount(wavH);
. . .
waveModCount2 = WaveModCount(wavH);
if (waveModCount2 != waveModCount1)
   // Wave has changed.
```

This routine was added in Igor Pro 4.0. If you are running with an earlier version of Igor, it will always return 0.

```
int
WaveModState(waveHandle)
waveHndl waveHandle;       // Handle to wave's data structure
```

Returns the truth that the wave has been modified since the last save to disk.

```
int
WaveLock(waveHandle)
waveHndl waveHandle;       // Handle to wave's data structure
```

Returns the lock state of the wave.

A return value of 0 signifies that the wave is not locked.

A return value of 1 signifies that the wave is locked. In that case, you should not kill the wave or modify it in any way.

Added in Igor Pro 5.00. When running with an earlier version, WaveLock always returns 0.

```
int
SetWaveLock(waveHandle, lockState)
waveHndl waveHandle;       // Handle to wave's data structure
int lockState;             // 0 or 1
```

Sets wave's lock state. If lockState is 0, the wave will be unlocked. If it is 1, the wave will be locked. All other bits are reserved.

Returns the previous state of the wave lock setting.

Added in Igor Pro 5.00. When running with an earlier version, SetWaveLock is a NOP and always returns 0.

```
void
WaveScaling(waveHandle, dxPtr, x0Ptr, topPtr, botPtr)
waveHndl waveHandle;       // Handle to wave's data structure
double *dxPtr, *x0Ptr, *topPtr, *botPtr;
```

Returns the wave's X and data scaling information.

Igor calculates the X value for point p of the wave as:

X value = x0 + dx*p

top and bottom are the values user entered for the wave's data full scale.

If both are zero then there is no data full scale for this wave.

New XOPs should use MDGetWaveScaling instead of WaveScaling so that they can access scaling for higher dimensions.

```
void
SetWaveScaling(waveHandle, dxPtr, x0Ptr, topPtr, botPtr)
waveHndl waveHandle;       // Handle to wave's data structure
double *dxPtr, *x0Ptr, *topPtr, *botPtr;
```

Sets the X scaling and data full scale for the specified wave.

If either dxPtr or x0Ptr is NULL, the X scaling is not set. Otherwise, Igor sets the X scaling of the wave. This is the same setting as is set by the SetScale x operation from within Igor. Igor calculates the X value for point p of the wave as:

X value = x0 + dx*p

If either topPtr or botPtr is NULL, the data full-scale is not set. Otherwise, Igor sets the data full scale of the wave. This is the same setting as is set by the SetScale d operation from within Igor. This setting is used only for documentation purposes. For example, if data was acquired on a -10 to 10 volt range, you would set the data full scale to (-10, 10).

New XOPs should use MDSetWaveScaling instead of SetWaveScaling so that they can access scaling for higher dimensions.

```
void
WaveUnits(waveHandle, xUnits, yUnits)
waveHndl waveHandle;        // Handle to wave's data structure
char* xUnits;               // C string defining wave's X units
char* dataUnits;            // C string defining wave's data units
```

Returns the wave's X and data units.

The units are strings like "kg", "m", or "s", or "" for no units.

In Igor Pro 3.0, the number of characters allowed was increased from 3 to 49 (MAX_UNIT_CHARS). For backward compatibility, WaveUnits will return no more than 3 characters (plus the null terminator). To get the full units and to access units for higher dimensions, new XOPs should use MDGetWaveUnits instead of WaveUnits.

```
void
SetWaveUnits(waveHandle, xUnits, yUnits)
waveHndl waveHandle;        // Handle to wave's data structure
char* xUnits;               // C string defining wave's X units
char* dataUnits;            // C string defining wave's data units
```

Sets the wave's X and data units.

The units are strings like "kg", "m", or "s", or "" for no units.

If xUnits is NULL the X units are not set.

If dataUnits is NULL, the data units are not set.

When running with Igor Pro 3.0 or later, units can be up to 49 (MAX_UNIT_CHARS) characters long.

In earlier versions of Igor, units are limited to 3 characters. You can pass a longer units string but only the first 3 characters will be used.

New XOPs should use MDSetWaveUnits instead of SetWaveUnits so that they can access units for higher dimensions.

```
Handle
WaveNote(waveHandle)
waveHndl waveHandle;        // Handle to wave's data structure
```

Returns a handle to the wave's note text or NULL if the wave has no wave note.

Note that the text in the handle is *not* null terminated. Use GetHandleSize to find the number of bytes in the handle. To use C string functions on this text you need to copy it to a local buffer and null-terminate it or add a null terminator to the handle and lock the handle. See **Understand the Difference Between a String in a Handle and a C String** on page 321.

This is the handle that Igor uses to store the wave's note and it belongs to Igor. Therefore, you should not modify or dispose of this handle. If you want to change the wave's note, use the SetWaveNote callback.

```
void
SetWaveNote(waveHandle, noteHandle)
waveHndl waveHandle;       // Handle to wave's data structure
Handle noteHandle;         // Handle to the text for the wave's note
```

Sets the wave's note.

noteHandle should be a handle to plain text with lines separated by carriage returns (ASCII 13) and with no null termination at the end. Once you pass the noteHandle to Igor it belongs to Igor so don't modify or dispose of it. This should be a handle that you created in your XOP using the NewHandle or HandToHand, not a handle you got from Igor.

If noteHandle is NULL the note is set to empty.

```
void
WaveName(waveHandle, namePtr)
waveHndl waveHandle;             // Handle to wave's data structure
char name[MAX_OBJ_NAME+1];       // C string to receive name
```

Puts the wave's name in the string identified by namePtr.

```
void*
WaveData(waveHandle)
waveHndl waveHandle;       // Handle to wave's data structure
```

Returns pointer to the start of wave's data.

**NOTE**: The pointer is only valid if the block of memory containing the wave's data is not relocated by the memory manager. Make sure that the handle is locked or that you do nothing that can cause it to be relocated while you use the pointer. See **Dangling Pointer / Heap Scramble Problems** on page 312 for details on locking and unlocking handles.

See Chapter 7 for detailed instructions on accessing wave data.

```
long
GetWavesInfo(waves,numWaves,waveTypes,wavePoints,waveStates,wavePtrs)
waveHndl waves[];           // Array of wave handles
int numWaves;               // Number of wave handles in waves[]
int waveTypes[];            // On output, array of wave numeric types
long wavePoints[];          // On output, array of wave lengths
int waveStates[];           // On output, array of waveHndl hStates
float* wavePtrs[];          // On output, array of pointers to wave data
```

Given an array of wave handles and the number of wave handles in the array, GetWavesInfo returns the type, length, handle state and pointer to the start of data for each wave via waveTypes, waveLengths, waveStates and wavePtrs. Each array must be able to hold at least numWaves elements.

If a given wave handle in waves is NULL, GetWavesInfo skips that element.

If waveStates is not NULL, it puts the current locked/unlocked state of the wave handle in the waveStates array, moves the handle to the top of the heap and locks it. If waveStates is NULL, it does none of this.

The function result is the length of the longest wave.

GetWavesInfo is of use when you need to lock a number of wave handles to access their data and then need to restore them to the state they were in before.

```
void
SetWavesStates(waves, numWaves, waveStates)
waveHndl waves[];           // Array of wave handles
int numWaves;               // Number of wave handles in waves[]
int waveStates[];           // On output, array of waveHandle hStates
```

Given an array of wave handles and the number of wave handles in the array, SetWavesStates sets the locked/unlocked state for each wave handle according to states in waveStates.

If a given wave handle in the waves array is NULL, SetWavesStates skips that element.

This call is the companion to GetWavesInfo. You would call this to restore wave handles after locking them and accessing their data.

```
void
WaveHandleModified(waveHandle)
waveHndl waveHandle;       // Handle to wave's data structure
```

Informs Igor that your XOP modified the specified wave.

At the time of the next update, Igor will update any windows that display the wave.

```
void
WaveHandlesModified(waves, numWaves, start, end)
waveHndl waves[];    // NumWaves handles to wave data
int numWaves;        // Number of wave handles in waves[]
long start[];        // Start point numbers for modified range or NULL
long end[];          // End point numbers for modified range or NULL
```

Informs Igor that your XOP modified the specified waves.

If any element of the waves array is NULL, Igor skips that element.

start and end are arrays of start and end point numbers for the modified range of each wave.

If start is NULL, the range start for each wave is the start of that wave.

If end is NULL, the range end for each wave is the end of that wave.

Currently, Igor updates the entire wave for each wave in the waves array if the wave is displayed in a graph or table. A future version of Igor may use the range information to minimize the amount of updating that needs to be done.

```
void
WaveModified(waveName)
char* waveName;            // C string containing wave's name
```

Informs Igor that your XOP modified the wave referred to by waveName in the current data folder.

So that your XOP will not depend on the current data folder, you should use WaveHandleModified instead of WaveModified.

```
int
MDMakeWave(wavHPtr,waveName,dataFolderH,dimensionSizes,type,overwrite)
waveHndl* wavHPtr;   // Place to put waveHndl for new wave
char* waveName;      // C string containing name of Igor wave
DataFolderHandle dataFolderH;         // Handle to data folder or NULL.
long dimensionSizes[MAX_DIMENSIONS+1]; // Array of dimension sizes
int type;            // Data type for new wave
int overwrite;       // If non-zero, overwrites existing wave
```

Makes a wave with the specified name, type and dimension sizes in the specified data folder.

If dataFolderH is NULL, it uses the current folder.

See MakeWave for a list of valid data types.

If overwrite is non-zero, an existing wave with the same name will be overwritten. However, as of this writing, Igor can not overwrite a text wave with a numeric wave or vice-versa and you will receive an error if you attempt to do this.

If overwrite is zero and a wave of the specified name exists, MDMakeWave will return a non-zero error code result.

For each dimension, dimensionSizes[i] specifies the number of elements in that dimension. For a wave of dimension n, i goes from 0 to n-1.

**NOTE**: dimensionSizes[n] must be zero. This is how Igor determines how many dimensions the wave is to have.

Returns error code or 0 if wave was made.

***Example***

```
waveHndl waveH;
char waveName[MAX_OBJ_NAME+1];
long dimensionSizes[MAX_DIMENSIONS+1];
int result;

strcpy(waveName, "Wave3D");
dimensionSizes[0] = 10;          // 10 rows
dimensionSizes[1] = 10;          // 10 columns
dimensionSizes[2] = 10;          // 10 layers
dimensionSizes[3] = 0;           // 0 marks first unused dimension
if (result = MDMakeWave(&waveH, waveName, NULL, dimensionSizes, NT_FP64, 1))
   return result;
```

```
int
MDGetWaveDimensions(wavH, numDimensionsPtr, dimensionSizes)
waveHndl wavH;              // Handle to the wave of interest
long* numDimensionsPtr;     // Number of dimensions in the wave
long dimensionSizes[MAX_DIMENSIONS+1]; // Array of dimension sizes
```

Returns the number of used dimensions in wave via numDimensionsPtr and the number of points in each used dimension via dimensionSizes.

If you only want to know the number of dimensions, you can pass NULL for dimensionSizes.

**NOTE**: dimensionSizes (if not NULL) should have room for MAX_DIMENSIONS+1 values.

For an n dimensional wave, MDGetWaveDimensions sets dimensionSizes[0..n-1] to the number of elements in the corresponding dimension and sets dimensionSizes[n..MAX_DIMENSIONS] to zero, indicating that they are unused dimensions. This guarantees that there will always be an element containing zero in the dimensionSizes array.

The function result is 0 or an Igor error code.

***Example***

```
long numDimensions;
long dimensionSizes[MAX_DIMENSIONS+1];
int result;

if (result = MDGetWaveDimensions(wavH, &numDimensions, dimensionSizes))
   return result;
```

```
int
MDChangeWave(wavH, dataType, dimensionSizes)
waveHndl wavH;               // Handle to the wave of interest
int dataType;                // New numeric type or -1 for no change
long dimensionSizes[MAX_DIMENSIONS+1]; // Array of new dimension sizes
```

Changes one or more of the following:

The wave's data type.

The number of dimensions in the wave.

The number of points in one or more dimensions.

dataType is one of the following:

-1 for no change in data type.

One of the data types listed for the MakeWave XOPSupport routine.

Except for TEXT_WAVE_TYPE, the data types may be ORed with NT_COMPLEX to make the wave complex.

Converting a text wave to numeric or vice versa is currently not supported and will result in an error.

dimensionSizes[i] contains the desired number of points for dimension i.

For n dimensions, dimensionSizes[n] must be zero. Then the size of each dimension is set by dimensionSizes[0..n-1]. If dimensionSizes[i] == -1, then the size of dimension i will be unchanged.

The function result is 0 or an error code.

***Example***

```
int dataType;
long dimensionSizes[MAX_DIMENSIONS+1];
int result;

// Clear all dimensions sizes to avoid undefined values.
MemClear(dimensionSizes, sizeof(dimensionSizes));
dimensionSizes[0] = 10;           // 10 rows
dimensionSizes[1] = 10;           // 10 columns
dimensionSizes[2] = 10;           // 10 layers
dimensionSizes[3] = 0;            // 0 marks first unused dimension
dataType = NT_FP64 | NT_CMPLX;    // complex, double-precision
if (result = MDChangeWave(wavH, dataType, dimensionSizes))
   return result;
```

If you are running with Igor version<3.0, attempts to make waves of more than one dimension will generate a non-zero error code result.

```
int
MDChangeWave2(wavH, dataType, dimensionSizes, mode)
waveHndl wavH;            // Handle to the wave of interest
int dataType;             // New numeric type or -1 for no change
long dimensionSizes[MAX_DIMENSIONS+1]; // Array of new dimension sizes
int mode;
```

This is the same as MDChangeWave except for the added mode parameter.

| Mode | Data Format |
|---|---|
| Mode=0 | Does a normal redimension. |
| Mode=1 | Changes the wave's dimensions without changing the wave data.<br>This is useful, for example, when you have a 2D wave consisting of 5 rows and 3 columns which you want to treat as a 2D wave consisting of 3 rows and 5 columns or if you have loaded floating point data into an unsigned byte wave. |
| Mode=2 | Changes the wave data from big-endian to little-endian or vice versa.<br>This is useful when you have loaded data from a file that uses a byte ordering different from that of the platform on which you are running. |

Returns 0 or an error code.

Added in Igor Pro 5.04B06. If mode is zero it will work with Igor Pro 3.0 or later. If mode is non-zero and the version of Igor is earlier than 5.04B06 it will return IGOR_OBSOLETE and do nothing.

```
int
MDGetWaveScaling(wavH, dimension, sfAPtr, sfBPtr)
waveHndl wavH;                  // Handle to the wave of interest
int dimension;
double* sfAPtr;                 // Delta value goes here
double* sfBPtr;                 // Offset value goes here
```

Returns the dimension scaling values or the data full scale values for the wave via sfAPtr and sfBPtr. If dimension is -1, it returns the data full scale values. Otherwise, it returns the dimension scaling for the specified dimension.

For dimension i (i=0 to 3), the scaled index for point p is:

<scaled index> = sfA[i]*p + sfB[i]

If dimension is -1, this gets the wave's data full scale setting instead of dimension scaling. *sfAPtr points to the top full scale value and *sfBPtr points to the bottom full scale value.

See **Wave Scaling and Units** on page 218 for a discussion of the distinction between dimension scaling and data full scale.

The function result is 0 or an Igor error code.

***Example***

```
double sfA;
double sfB;
int result;

if (result = MDGetWaveScaling(wavH, ROWS, &sfA, &sfB))       // Get X scaling
   return result;
if (result = MDGetWaveScaling(wavH, COLUMNS, &sfA, &sfB))    // Get Y scaling
   return result;
```

```
int
MDSetWaveScaling(wavH, dimension, sfAPtr, sfBPtr)
waveHndl wavH;                      // Handle to the wave of interest
int dimension;
double* sfAPtr;                     // Points to new delta value
double* sfBPtr;                     // Points to new offset value
```

Sets the dimension scaling values or the data full scale values for the wave via sfAPtr and sfBPtr. If dimension is -1, it sets the data full scale values. Otherwise, it sets the dimension scaling for the specified dimension.

For dimension i (i=0 to 3), the scaled index of point p is:

<scaled index> = sfA[i]*p + sfB[i]

If dimension is -1, this sets the wave's data full scale setting instead of dimension scaling. *sfAPtr points to the top full scale value and *sfBPtr points to the bottom full scale value.

See **Wave Scaling and Units** on page 218 for a discussion of the distinction between dimension scaling and data full scale.

The function result is 0 or an Igor error code.

***Example***

This example sets the scaling of the row and column dimensions to the default (scaled value == row/column number) without affecting the scaling of any other dimensions that might exist in the wave.

```
double sfA;
double sfB;
```

```
int result;

sfA = 1.0; sfB = 0.0;
if (result= MDSetWaveScaling(wavH, ROWS, &sfA, &sfB))    // Set X scaling
   return result;
sfA = 1.0; sfB = 0.0;
if (result= MDSetWaveScaling(wavH, COLUMNS, &sfA, &sfB)) // Set Y scaling
   return result;
```

```
int
MDGetWaveUnits(wavH, dimension, units)
waveHndl wavH;                      // Handle to the wave of interest
int dimension;
char units[MAX_UNIT_CHARS+1];       // C string
```

Returns the units string for a dimension in the wave or for the wave's data via units. If dimension is -1, it gets the data units. Otherwise, it gets the dimension units for the specified dimension (ROWS, COLUMNS, LAYERS, CHUNKS).

The function result is 0 or an Igor error code.

Units may be up to 49 (MAX_UNIT_CHARS) characters. You should allocate MAX_UNIT_CHARS+1 bytes for units.

See **Wave Scaling and Units** on page 218 for a discussion of the distinction between dimension units and data units.

***Example***

```
long numDimensions;
int dimension;
char units[MAX_UNIT_CHARS+1];
char buf[256];
int result;

if (result = MDGetWaveDimensions(wavH, &numDimensions, NULL))
   return result;
for (dimension=0; dimension<numDimensions; dimension++) {
   if (result = MDGetWaveUnits(wavH, dimension, units))
      return result;
   sprintf(buf, "Units for dimension %d: \"%s\""CR_STR, dimension, units);
   XOPNotice(buf);
}
```

```
int
MDSetWaveUnits(wavH, dimension, units)
waveHndl wavH;                    // Handle to the wave of interest
int dimension;
char units[MAX_UNIT_CHARS+1];     // C string
```

Sets the units string for a dimension in the wave or for the wave's data via units. If dimension is -1, it sets the data units. Otherwise, it sets the dimension units for the specified dimension (ROWS, COLUMNS, LAYERS, CHUNKS).

The function result is 0 or an Igor error code.

Units may be up to 49 (MAX_UNIT_CHARS) characters. If the string you pass is too long, Igor will store a truncated version of it.

See **Wave Scaling and Units** on page 218 for a discussion of the distinction between dimension units and data units.

***Example***

```
char units[MAX_UNIT_CHARS+1];
int result;

if (result = MDSetWaveUnits(wavH, 0, "s"))    // Set X units to seconds
   return result;
if (result = MDSetWaveUnits(wavH, -1, "v"))   // Set data units to volts
   return result;
```

```
int
MDGetDimensionLabel(wavH, dimension, element, dimLabel)
waveHndl wavH;              // Handle to the wave of interest
int dimension;              // The dimension of interest.
long element;               // The element whose label is to be gotten.
char* dimLabel;             // Label returned here.
```

Returns the label for the specified element of the specified dimension as a C string via dimLabel.

You should allocate dimLabel as follows:

```
char dimLabel[MAX_DIM_LABEL_CHARS+1];
```

MAX_DIM_LABEL_CHARS is defined in IgorXOP.h to be 255. At present, Igor truncates labels at 31 characters. This may be increased up to 255 in a future version without requiring your XOP to be recompiled.

If element is -1, this specifies a label for the entire dimension. If element is between 0 and n-1, where n is the size of the dimension, then element specifies a label for that element of the dimension only. You will receive an error if dimension or element is less than -1 or greater than n-1.

A dimension label may be empty ("").

The function result is 0 or an Igor error code.

See the WaveAccess sample XOP for an example.

```
int
MDSetDimensionLabel(wavH, dimension, element, dimLabel)
waveHndl wavH;              // Handle to the wave of interest
int dimension;              // The dimension of interest
long element;               // The element whose label is to be gotten.
char* dimLabel;             // You pass value here
```

Sets the label for the specified element of the specified dimension.

At present, Igor will truncate the label at 31 characters. This may be increased in the future without requiring your XOP to be recompiled.

If element is -1, this specifies a label for the entire dimension. If element is between 0 and n-1, where n is the size of the dimension, then element specifies a label for that element of the dimension only. You will receive an error if dimension or element is less than -1 or greater than n-1.

The function result is 0 or an Igor error code.

See the WaveAccess sample XOP for an example.

```
int
GetWaveDimensionLabels(waveH, dimLabelsHArray)
waveHndl waveH;                          // Handle to the wave of
interest
Handle dimLabelsHArray[MAX_DIMENSIONS];   // Labels returned via this
array
```

dimLabelsHArray points to an array of MAX_DIMENSIONS handles. GetWaveDimensionLabels sets each element of this array to a handle containing dimension labels or to NULL.

On output, if the function result is 0 (no error), dimLabelsHArray[i] will be a handle containing dimension labels for dimension i or NULL if dimension i has no dimension labels.

If the function result is non-zero then all handles in dimLabelsHArray will be NULL.

Any non-NULL output handles belong to you. Dispose of them with DisposeHandle when you are finished with them.

For each dimension, the corresponding dimension label handle consists of an array of N+1 C strings, each in a field of (MAX_DIM_LABEL_CHARS+1) bytes.

The first label is the overall dimension label for that dimension.

Label i+1 is the dimension label for element i of the dimension.

N is the smallest number such that the last non-empty dimension label for a given dimension and all dimension labels before it, whether empty or not, can be stored in the handle.

For example, if a 5 point 1D wave has dimension labels for rows 0 and 2 with all other dimension labels being empty then dimLabelsHArray[0] will contain four dimension labels, one for the overall dimension and three for rows 0 through 2. dimLabelsHArray[0] will not contain any storage for any point after row 2 because the remaining dimension labels for that dimension are empty.

Returns 0 or an error code.

For an example using this routine, see TestGetAndSetWaveDimensionLabels in XOPWaveAccess.c.

Added for Igor Pro 5.04. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

```
int
SetWaveDimensionLabels(waveH, dimLabelsHArray)
waveHndl waveH;                            // Handle to the wave of
interest
Handle dimLabelsHArray[MAX_DIMENSIONS];    // Labels passed in this array
```

dimLabelsHArray points to an array of MAX_DIMENSIONS handles. SetWaveDimensionLabels sets the dimension labels for each existing dimension of waveH based on the corresponding handle in dimLabelsHArray.

The handles in dimLabelsHArray belong to you. Dispose of them with DisposeHandle when you are finished with them.

See the documentation for GetWaveDimensionLabels for a discussion of how the dimension labels are stored in the handles.

Returns 0 or an error code.

For an example using this routine, see TestGetAndSetWaveDimensionLabels in XOPWaveAccess.c.

Added for Igor Pro 5.04. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

```
int
MDAccessNumericWaveData(wavH, accessMode, dataOffsetPtr)
waveHndl wavH;                  // Handle to the wave of interest
int accessMode;
long* dataOffsetPtr;
```

MDAccessNumericWaveData provides one of several methods of access to the data for numeric waves. MDAccessNumericWaveData is the fastest of the access methods but also is the most difficult to use. See **Accessing Numeric Wave Data** on page 212 for a comparison of the various methods.

wavH is the wave handle containing the data you want to access.

accessMode is a code that tells Igor how you plan to access the wave data and is used for a future compatibility check. At present, there is only one accessMode. You should use the symbol kMDWaveAccessMode0 for the accessMode parameter.

On output, if there is no error, *dataOffsetPtr contains the offset in bytes from the start of the wave handle to the data. You can use this offset to point to the start of the wave data in the wave handle. See **Accessing Numeric Wave Data** on page 212 for a discussion of how wave data is layed out in memory.

The function result is 0 or an error code.

If it returns a non-zero error code, you should not attempt to access the wave data but merely return the error code to Igor as the result of your function or operation. At present, there is only one case in which MDAccessNumericWaveData will return an error code – if the wave is a text wave.

It is possible, though unlikely, that a future version of Igor Pro will store wave data in a different way, such that the current method of accessing wave data will no longer work. If your XOP ever runs with such a future Igor, MDAccessNumericWaveData will return an error code indicating the incompatibility. Your XOP will refrain from attempting to access the wave data and return the error code to Igor. This will prevent a crash and indicate the nature of the problem to the user.

***Example***

This example adds one to each element of a wave of two dimensions. It illustrates the difficulty in using MDAccessNumericWaveData - you need to take into account the data type of the wave that you are accessing. The other access methods, described in Chapter 7, don’t require this.

```
long numDimensions;
long dimensionSizes[MAX_DIMENSIONS+1];
long numRows, numColumns, row, column, dataOffset;
int type, type2, isComplex;
char* cp; short* sp; long* lp;
unsigned char* ucp; unsigned short* usp; unsigned long* ulp;
float* fp; double* dp;
```

```
int hState, result;

type = WaveType(wavH);
type2 = type & ~NT_CMPLX;        // Type without the complex bit set
isComplex = type & NT_CMPLX;
if (type2 == TEXT_WAVE_TYPE)
   return NUMERIC_ACCESS_ON_TEXT_WAVE;
if (result = MDGetWaveDimensions(wavH, &numDimensions, dimensionSizes))
   return result;
if (numDimensions != 2)
   return REQUIRES_2D_WAVE;      // An error code defined by your XOP
numRows = dimensionSizes[ROWS];
numColumns = dimensionSizes[COLUMNS];
if (result=MDAccessNumericWaveData(wavH,kMDWaveAccessMode0,&dataOffset))
   return result;
hState = MoveLockHandle(wavH);   // Lock handle so data won't move.
dp= (double*)((char*)(*wavH) + dataOffset); fp=(float*)dp;
lp= (long*)dp; sp=(short*)dp; cp=(char*)dp;
ulp=(unsigned long*)dp; usp=(unsigned short*)dp;
ucp=(unsigned char*)dp;
for(column=0; column<numColumns; column++) {
   for(row=0; row<numRows; row++) {
      switch(type2) {
         case NT_FP64:
            *dp++ += 1; if (isComplex) *dp++ += 1; break;
         case NT_FP32:
            *fp++ += 1; if (isComplex) *fp++ += 1; break;
         case NT_I32:
            *lp++ += 1; if (isComplex) *lp++ += 1; break;
         case NT_I16:
            *sp++ += 1; if (isComplex) *sp++ += 1; break;
         case NT_I8:
            *cp++ += 1; if (isComplex) *cp++ += 1; break;
         case NT_I32 | NT_UNSIGNED:
            *ulp++ += 1; if (isComplex) *ulp++ += 1; break;
         case NT_I16 | NT_UNSIGNED:
            *usp++ += 1; if (isComplex) *usp++ += 1; break;
         case NT_I8 | NT_UNSIGNED:
            *ucp++ += 1; if (isComplex) *ucp++ += 1; break;
         default:    // Unknown data type - possible in a future Igor.
            HSetState((Handle)wavH, hState);
            return NT_FNOT_AVAIL;   // Func not supported on this type.
      }
   }
}
HSetState((Handle)wavH, hState);
```

```
int
MDGetNumericWavePointValue(wavH, indices, value)
waveHndl wavH;                    // Handle to the wave of interest
long indices[MAX_DIMENSIONS];     // Identifies the point of interest
double value[2];                  // Value returned here
```

Returns via value the value of a particular element in the specified numeric wave.

The value returned is always double precision floating point, regardless of the precision of the wave.

indices is an array of dimension indices. For example, for a 3D wave:

indices[0] contains the row number

indices[1] contains the column number

indices[2] contains the layer number

This routine ignores indices for dimensions that do not exist in the wave.

The real part of the value of specified point is returned in value[0].

If the wave is complex, the imaginary part of the value of specified point is returned in value[1]. If the wave is not complex, value[1] is undefined.

The function result is 0 or an error code.

Currently the only error code returned is MD_WAVE_BAD_INDEX, indicating that you have passed one or more invalid indices. An index for a particular dimension is invalid if it is less than zero or greater than or equal to the number of points in the dimension. Future versions of Igor may return other error codes. If you receive an error, just return it to Igor so that it will be reported to the user.

See the example below for MDSetNumericWavePointValue.

```
int
MDSetNumericWavePointValue(wavH, indices, value)
waveHndl wavH;                    // Handle to the wave of interest
long indices[MAX_DIMENSIONS];     // Identifies the point of interest
double value[2];                  // Value returned here
```

Sets the value of a particular point in the specified numeric wave.

The value that you supply is always double precision floating point, regardless of the precision of the wave.

indices is an array of dimension indices. For example, for a 3D wave:

indices[0] contains the row number

indices[1] contains the column number

indices[2] contains the layer number

This routine ignores indices for dimensions that do not exist in the wave.

You pass in value[0] the real part of the value.

If the wave is complex, you pass the imaginary part in value[1]. If the wave is not complex, MDSetNumericWavePointValue ignores value[1].

When storing into an integer wave, MDSetNumericWavePointValue truncates the value that you are storing. If you want, you can do rounding before calling MDSetNumericWavePointValue.

The function result is 0 or an error code.

Currently the only error code returned is MD_WAVE_BAD_INDEX, indicating that you have passed one or more invalid indices. An index for a particular dimension is invalid if it is less than zero or greater than or equal to the number of points in the dimension. Future versions of Igor may return other error codes. If you receive an error, just return it to Igor so that it will be reported to the user.

***Example***

This example adds one to each element of a wave of two dimensions. It illustrates the ease in using MDGetNumericWavePointValue and MDSetNumericWavePointValue – you don't need to user pointers and you don't need to take into account the data type of the wave that you are accessing. However, it is somewhat slower than the other methods. See Chapter 7 for speed comparisons.

```
long numDimensions;
long dimensionSizes[MAX_DIMENSIONS+1];
long numRows, numColumns;
long row, column;
long indices[MAX_DIMENSIONS];
int isComplex;
```

```
double value[2];
int result;

isComplex = WaveType(wavH) & NT_CMPLX;

if (result = MDGetWaveDimensions(wavH, &numDimensions, dimensionSizes))
   return result;
if (numDimensions != 2)
   return REQUIRES_2D_WAVE;          // An error code defined by your XOP
numRows = dimensionSizes[0];
numColumns = dimensionSizes[1];

MemClear(indices, sizeof(indices)); // Must be 0 for unused dimensions.
for(column=0; column<numColumns; column++) {
   indices[1] = column;
   for(row=0; row<numRows; row++) {
      indices[0] = row;
      if (result = MDGetNumericWavePointValue(wavH, indices, value))
         return result;
      value[0] += 1;              // Real part
      if (isComplex)
         value[1] += 1;           // Imag part
      if (result = MDSetNumericWavePointValue(wavH, indices, value))
         return result;
   }
}
```

```
int
MDGetDPDataFromNumericWave(wavH, dPtr)
waveHndl wavH;                    // Handle to the wave of interest
double* dPtr;                     // Where to put the data
```

MDGetDPDataFromNumericWave stores a double-precision copy of the specified numeric wave's data in the memory pointed to by dPtr. dPtr must point to a block of memory that you have allocated and which must be at least (WavePoints(wavH)*sizeof(double)) bytes or twice that for a complex wave.

This routine is a companion to MDStoreDPDataInNumericWave.

The function result is zero or an error code.

See the example below for MDStoreDPDataInNumericWave.

```
int
MDStoreDPDataInNumericWave(wavH, dPtr)
waveHndl wavH;                      // Handle to the wave of interest
double* dPtr;                       // Pointer to the data to store
```

MDStoreDPDataInNumericWave stores the data pointed to by dPtr in the specified numeric wave.

During the transfer, it converts the data from double precision to the numeric type of the wave. The conversion is done on-the-fly and the data pointed to by dPtr is not changed.

When storing into an integer wave, MDStoreDPDataInNumericWave truncates the value that you are storing. If you want, you can do rounding before calling MDStoreDPDataInNumericWave.

This routine is a companion to MDGetDPDataInNumericWave.

The function result is zero or an error code.

Numeric wave data is stored contiguously in the wave handle in one of the supported data types (see the MakeWave routine). To access the a particular point, you need to know the number of data points in each dimension. To find this, you must call MDGetWaveDimensions. This returns the number of used dimensions in the wave and an array of dimension lengths. The dimension lengths are interpreted as follows:

| | |
|---|---|
| dimensionSizes[ROWS] | Number of rows in a column |
| dimensionSizes[COLUMNS] | Number of columns in a layer |
| dimensionSizes[LAYERS] | Number of layers in a chunk |
| dimensionSizes[CHUNKS] | Number of chunks in the wave |

ROWS, COLUMNS, LAYERS and CHUNKS are defined in IgorXOP.h as 0, 1, 2 and 3.

The data is stored in row/column/layer/chunk order. This means that, as you step linearly through memory one point at a time, you first pass the value for each row in the first column. At the end of the first column, you reach the start of the second column. After you have passed the data for each column in the first layer, you reach the data for the first column in the second layer. After you have passed the data for each layer, you reach the data for the first layer of the second chunk. And so on.

***Example***

This example adds one to each element of a wave of two dimensions. It illustrates the ease in using MDGetDPDataFromNumericWave and MDStoreDPDataInNumericWave – you don't need to take into account the data type of the wave that you are accessing. However, it requires that you make a copy of the wave data which requires more memory than the other methods.

```
long numDimensions;
long dimensionSizes[MAX_DIMENSIONS+1];
```

```
long numRows, numColumns, numBytes;
long row, column;
int isComplex;
double* dPtr;
double* dp;
int result;

isComplex = WaveType(wavH) & NT_CMPLX;

if (result = MDGetWaveDimensions(wavH, &numDimensions, dimensionSizes))
   return result;
if (numDimensions != 2)
   return REQUIRES_2D_WAVE;      // an error code defined by your XOP
numRows = dimensionSizes[0];
numColumns = dimensionSizes[1];

numBytes = WavePoints(wavH) * sizeof(double);   // bytes needed for copy
if (isComplex)
   numBytes *= 2;
dPtr = (double*)NewPtr(numBytes);
if (dPtr==NULL)
   return NOMEM;

if (result = MDGetDPDataFromNumericWave(wavH, dPtr)) {
   DisposePtr((Ptr)dPtr);
   return result;
}

dp = dPtr;
for(column=0; column<numColumns; column++) {
   for(row=0; row<numRows; row++) {
      *dp++ += 1;                // real part
      if (isComplex)
         *dp++ += 1;             // imag part
   }
}

if (result = MDStoreDPDataInNumericWave(wavH, dPtr)) {
   DisposePtr((Ptr)dPtr);
   return result;
}

DisposePtr((Ptr)dPtr);
```

```
int
FetchNumericValue(type, dataStartPtr, index, value)
int type;                          // Igor numeric data type
char* dataStartPtr;                // Pointer to start of wave data
long index;                        // Point number index
double value[2];                   // Point value is returned here
```

Returns the value of one element of data of the specified type. The returned value is always double precision floating point.

type is an Igor numeric type (see the MakeWave routine).

dataStartPtr points to the start of the numeric data.

index is an index in point numbers from dataStartPtr to the point of interest.

The real part of the value of specified point is returned in value[0].

If the data is complex, the imaginary part of the value of specified point is returned in value[1]. If the data is not complex, value[1] is undefined.

FetchNumericValue is a low-level routine used by MDGetNumericWavePointValue. It treats the wave as a linear array, ignoring its dimensionality. Normally, you will have no need to use it directly. However, advanced users may find it useful. It has the ability to convert from any Igor numeric data type to double precision.

This routine is a companion to StoreNumericValue.

The function result is zero or an error code.

```
int
StoreNumericValue(type, dataStartPtr, index, value)
int type;                          // Igor numeric data type
char* dataStartPtr;                // Pointer to start of wave data
long index;                        // Point number index
double value[2];                   // Point value to be stored
```

Stores the specified value using the specified numeric type.

type is an Igor numeric type (see the MakeWave routine).

dataStartPtr points to the start of the numeric data.

index is an index in point numbers from dataStartPtr to the point of interest.

You should pass in value[0] the real part of the value.

If the data is complex, you should pass the complex part in value[1]. If the data is not complex, StoreNumericValue ignores value[1].

StoreNumericValue is a low-level routine used by MDSetNumericWavePointValue. It treats the wave as a linear array, ignoring its dimensionality. Normally, you will have no need to use it directly. However, advanced users may find it useful. It has the ability to convert from any Igor numeric data type to double precision.

This routine is a companion to FetchNumericValue.

The function result is zero or an error code.

```
int
MDGetTextWavePointValue(wavH, indices, textH)
waveHndl wavH;                    // Handle to the wave of interest
long indices[MAX_DIMENSIONS];     // Identifies the point of interest
Handle textH;                     // Value returned here
```

Returns via textH the value of a particular element in the specified wave.

The value is returned in the textH handle, which you must allocate before calling MDGetTextWavePointValue. Any previous contents of textH are overwritten. textH is yours to dispose when you are finished with it.

If the wave is not a text wave, returns an error code and does not alter textH.

indices is an array of dimension indices. For example, for a 3D wave:

indices[0] should contain the row number

indices[1] should contain the column number

indices[2] should contain the layer number

This routine ignores indices for dimensions that do not exist in the wave.

The function result is 0 or an error code.

On output, if there is no error, textH contains a copy of the characters in the specified wave element. An element in an Igor text wave can contain any number of characters, including zero. Therefore, the handle can contain any number of characters. Igor text waves can contain any characters, including control characters. No characters codes are considered illegal.

Note that the text in the handle is *not* null terminated. Use GetHandleSize to find the number of bytes in the handle. To use C string functions on this text you need to copy it to a local buffer and null-terminate it or add a null terminator to the handle and lock the handle. See **Understand the Difference Between a String in a Handle and a C String** on page 321. If you pass the handle back to Igor, you must remove the null terminator and unlock the handle.

See the example below for MDSetTextWavePointValue.

```
int
MDSetTextWavePointValue(wavH, indices, textH)
waveHndl wavH;                  // Handle to the wave of interest
long indices[MAX_DIMENSIONS];   // Identifies the point of interest
Handle textH;                   // Value to store in wave
```

Transfers the characters in textH to the specified point in the specified wave. The contents of textH is not altered.

If the wave is not a text wave, returns an error code.

indices is an array of dimension indices. For example, for a 3D wave:

indices[0] should contain the row number

indices[1] should contain the column number

indices[2] should contain the layer number

This routine ignores indices for dimensions that do not exist in the wave.

A point in an Igor text wave can contain any number of characters, including zero. Therefore, the handle can contain any number of characters. Igor text waves can contain any characters, including control characters. No characters codes are considered illegal.

The text in the handle must not be null terminated. If you have put a null terminator in the handle, remove it before calling MDSetTextWavePointValue.

After calling MDSetTextWavePointValue, the handle is still yours so you should dispose it when you no longer need it.

The function result is 0 or an error code.

***Example***

This example adds an asterisk to each element of a wave of two dimensions. It illustrates the ease of using MDGetTextWavePointValue and MDSetTextWavePointValue.

```
long numDimensions;
long dimensionSizes[MAX_DIMENSIONS+1];
long numRows, numColumns;
long row, column;
long indices[MAX_DIMENSIONS];
Handle textH;
int result;

if (result = MDGetWaveDimensions(wavH, &numDimensions, dimensionSizes))
   return result;
if (numDimensions != 2)
   return REQUIRES_2D_WAVE;      // an error code defined by your XOP
numRows = dimensionSizes[ROWS];
```

```
numColumns = dimensionSizes[COLUMNS];
textH = NewHandle(0L);
if (textH == NULL)
   return NOMEM;

MemClear(indices, sizeof(indices));
for(column=0; column<numColumns; column++) {
   indices[1] = column;
   for(row=0; row<numRows; row++) {
      indices[0] = row;
      if (result = MDGetTextWavePointValue(wavH, indices, textH))
         break;
      if (PtrAndHand("*", textH, 1)) {
         result = NOMEM;
         break;
      }
      if (result = MDSetTextWavePointValue(wavH, indices, textH))
         break;
   }
}

DisposeHandle(textH);
```

```
int
GetTextWaveData(waveH, mode, textDataHPtr)
waveHndl waveH;             // Handle to the wave of interest
int mode;                   // Determines format of returned data
Handle* textDataHPtr;       // Data is returned here
```

Returns all of the text for the specified text wave via textDataHPtr.

**NOTE**: This routine is for advanced programmers who are comfortable with pointer arithmetic and handles. Less experienced programmers should use MDGetTextWavePointValue to get the wave text values one at a time.

If the function result is 0 then *textDataHPtr is a handle that you own. When you are finished, dispose of it using DisposeHandle.

In the event of an error, the function result will be non-zero and *textDataHPtr will be NULL.

The returned handle will contain the text for all of the wave's elements in one of several formats explained below. The format depends on the mode parameter.

Modes 0 and 1 use a null byte to mark the end of a string and thus will not work if 0 is considered to be a legal character value.

**Mode** — **Data Format**

**Mode=0**

The returned handle contains one C string (null-terminated) for each element of the wave.

Example:

```
"Zero"<null>
"One"<null>
"Two"<null>
```

**Mode=1**

The returned handle contains a list of 32-bit offsets to strings followed by the string data. There is one extra offset which is the offset to where the string would be for the next element if the wave had one more element.

The text for each element in the wave is represented by a C string (null-terminated).

Example:

```
<Offset to "Zero">
<Offset to "One">
<Offset to "Two">
<Extra offset>
"Zero"<null>
"One"<null>
"Two"<null>
```

**Mode=2**

The returned handle contains a list of 32-bit offsets to strings followed by the string data.

The text for each element in the wave is not null-terminated.

Example:

```
<Offset to "Zero">
<Offset to "One">
<Offset to "Two">
<Extra offset>
"Zero"
"One"
"Two"
```

Using modes 1 and 2, you can determine the length of element i by subtracting offset i from offset i+1.

You can convert the offsets into pointers to strings by adding **textDataHPtr to each of the offsets. However, since the handle in theory can be relocated in memory, you should lock the handle before converting to pointers and unlock it when you are done with it.

For the purposes of GetTextWaveData, the wave is treated as a 1D wave regardless of its true dimensionality. If waveH a 2D text wave, the data returned via textDataHPtr is in column-major order. This means that the data for each row of the first column appears first in memory, followed by the data for the each row of the next column, and so on.

Returns 0 or an error code.

For an example using this routine, see TestGetAndSetTextWaveData in XOPWaveAccess.c.

Added for Igor Pro 5.04. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

```
int
SetTextWaveData(waveH, mode, textDataHPtr)
waveHndl waveH;             // Handle to the wave of interest
int mode;                   // Determines format of returned data
Handle textDataH;           // Data is passed to Igor here
```

Sets all of the text for the specified text wave according to textDataH.

**NOTE**: This routine is for advanced programmers who are comfortable with pointer arithmetic and handles. Less experienced programmers should use MDSetTextWavePointValue to get the wave text values one at a time.

**WARNING**: If you pass inconsistent data in textDataH you will cause Igor to crash.

SetTextWaveData can not change the number of points in the text wave. Therefore the data in textDataH must be consistent with the number of points in waveH. Otherwise a crash will occur.

Also, when using modes 1 or 2, the offsets must be correct. Otherwise a crash will occur.

Crashes caused by inconsistent data may occur at unpredictable times making it hard to trace it to the problem. So triple-check your code.

You own the textDataH handle. When you are finished with it, dispose of it using DisposeHandle.

The format of textDataH depends on the mode parameter. See the documentation for GetTextWaveData for a description of these formats.

Modes 0 and 1 use a null byte to mark the end of a string and thus will not work if 0 is considered to be a legal character value.

Returns 0 or an error code.

For an example using this routine, see TestGetAndSetTextWaveData in XOPWaveAccess.c.

Added for Igor Pro 5.04. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

# Routines for Accessing Variables

These routines allow you to create or get or set the value of Igor numeric and string variables.

Most of the routines listed in this section deal with global variables in the current data folder. The exceptions are GetNVAR, SetNVAR, GetSVAR and SetSVAR which are used in conjunction with fields in Igor Pro structures.

In addition to the routines in this section, you can use GetDataFolderObject and SetDataFolderObject to access global variables.

The routines in this section can also be used to create, set and read macro-local variables. Since macro programming is deprecated, we recommend that you use these routines to deal with global variables only. Where appropriate the routines have a parameter that allows you to specify that you want to deal with global variables.

Routines that deal with variables used by Operation Handler-based external operations are described in the section **Operation Handler Routines** on page 329.

To set file loader output variables, see **Routines for File-Loader XOPs** on page 426.

```
int
Variable(varName, varType)
char* varName;        // C string containing name of variable
int varType;          // Data type of variable
```

Creates an Igor numeric or string variable in the current data folder.

The function result is 0 if it was able to make the variable or an Igor error code if not.

If varType is 0, Igor makes a string variable. If it is NT_FP64, it makes a numeric variable. If it is (NT_FP64 | NT_CMPLX), it makes a complex numeric variable.

Variable can not create a local variable in a user-defined function. However, it can create a local variable in a macro. Since it is usually undesirable for your XOP to behave differently depending on how it is called, you should generally avoid creating macro local variables.

If a macro is running when Variable is called, the variable that is created is local to that macro. You can force the variable to be global by using the VAR_GLOBAL flag which is defined in IgorXOP.h. For example:

```
static int
Test(void)
{
   // These will always create global variables even if called from a macro.
```

```
   Variable("globalNumericVar", NT_FP64 | VAR_GLOBAL);
   Variable("globalStringVar", 0 | VAR_GLOBAL);
}
```

Prior to Igor Pro 5, the Variable function was used to create output variables from external operations, analogous to V_flag or S_fileName. When programming for Igor Pro 5 or later, you should use Operation Handler (see page 151) to implement your external operation. Operation Handler will create output variables for you so you should not use Variable for that purpose.

```
int
FetchNumVar(varName, doublePtr1, doublePtr2)
char* varName;              // C string containing name of variable
double* doublePtr1;         // Receives contents of real part of variable
double* doublePtr2;         // Receives contents of imag part of variable
```

Returns the value of the named Igor numeric variable in the current data folder.

Returns -1 if the variable does not exist or the numeric type of the variable if it does.

The numeric type will be either NT_FP64 (double-precision) or (NT_FP64 | NT_CMPLX) (double-precision, complex).

If the variable is not complex, the value returned through doublePtr2 is undefined but you still must pass in a pointer to a valid double because some versions of Igor will access it even if the Igor variable is not complex.

```
int
StoreNumVar(varName, doublePtr1, doublePtr2)
char* varName;              // C string containing name of variable
double* doublePtr1;         // Points to value to store in real part
double* doublePtr2;         // Points to value to store in imag
```

Sets the value of the named Igor numeric variable in the current data folder.

Returns -1 if the variable does not exist or the numeric type of the variable if it does.

The numeric type will be either NT_FP64 (double-precision) or (NT_FP64 | NT_CMPLX) (double-precision, complex).

If the variable is not complex, the value pointed to by doublePtr2 does not matter but it must point to a valid double variable because some versions of Igor will access it even if the Igor variable is not complex.

```
int
FetchStrVar(varName, stringPtr)
char* varName;              // C string containing name of string
char* stringPtr;            // Receives contents of string
```

Fetches the contents of the named Igor string variable in the current data folder.

The function result is 0 if it was able to fetch the string or an Igor error code if the string variable does not exist.

stringPtr should point to an array of 256 characters.

FetchStrVar will never return more than 255 characters. If you want to access any characters beyond the first 255, use FetchStrHandle instead of FetchStrVar.

```
int
StoreStrVar(varName, stringPtr)
char* varName;              // C string containing name of string
char* stringPtr;            // C string to store in string variable
```

Sets the contents of the named Igor string variable in the current data folder.

The function result is 0 if it was able to store the string or an Igor error code if the string variable does not exist.

There is no limit to the length of the C string pointed to by stringPtr.

```
Handle
FetchStrHandle(varName)
char* varName;              // C string containing name of string
```

Returns the handle containing the text for the named Igor string variable in the current data folder.

Returns NULL if there is no such string variable.

Note that the text in the handle is *not* null terminated. Use GetHandleSize to find the number of bytes in the handle. To use C string functions on this text you need to copy it to a local buffer and null-terminate it or add a null terminator to the handle and lock the handle. See **Understand the Difference Between a String in a Handle and a C String** on page 321.

You must not dispose of or otherwise modify this handle since it belongs to Igor.

You must use the handle immediately and then not refer to it again since Igor will dispose it if the user kills the string variable.

```
int
SetIgorIntVar(numVarName, value, forceGlobal)
char* numVarName;          // Name of the Igor numeric variable to set
long value;                // Value to set it to
int forceGlobal;           // Non-zero to create a global variable
```

Sets an Igor double-precision numeric variable in the current data folder to an integer value after creating it, if necessary.

The function result is 0 or an error code.

If the named numeric variable already exists, SetIgorIntVar just sets it. If it does not exist, it creates it and then sets it. In this case, it will create a global variable if forceGlobal is non-zero. If forceGlobal is zero, the variable will be local if a macro is running or global if not.

SetIgorIntVar can not create a local variable in a user-defined function. Since it is usually undesirable for your XOP to behave differently depending on how it is called, you should generally avoid creating macro local variables.

In the event of a name conflict with a wave or string variable in the same scope, SetIgorIntVar will return an error.

Prior to Igor Pro 5, the SetIgorIntVar function was used to create output variables from external operations, analogous to V_flag. When programming for Igor Pro 5 or later, you should use Operation Handler (see page 151) to implement your external operation. Operation Handler will create output variables for you so you should not use SetIgorIntVar for that purpose.

```
int
SetIgorFloatingVar(numVarName, valuePtr, forceGlobal)
char* numVarName;           // Name of the Igor numeric variable to set
double* valuePtr;           // Pointer to double value to set it to
int forceGlobal;            // Non-zero to create a global variable
```

Sets an Igor double-precision numeric variable in the current data folder to a floating-point value after creating it, if necessary.

The function result is 0 or an error code.

If the named numeric variable already exists, SetIgorFloatingVar just sets it. If it does not exist, it creates it and then sets it. In this case, it will create a global variable if forceGlobal is non-zero. If forceGlobal is zero, the variable will be local if a macro is running or global if not.

SetIgorFloatingVar can not create a local variable in a user-defined function. Since it is usually undesirable for your XOP to behave differently depending on how it is called, you should generally avoid creating macro local variables.

In the event of a name conflict with a wave or string variable in the same scope, SetIgorFloatingVar will return an error.

Prior to Igor Pro 5, the SetIgorFloatingVar function was used to create output variables from external operations, analogous to V_flag. When programming for Igor Pro 5 or later, you should use Operation Handler (see page 151) to implement your external operation. Operation Handler will create output variables for you so you should not use SetIgorFloatingVar for that purpose.

```
int
SetIgorComplexVar(numVarName, realPtr, imagPtr, forceGlobal)
char* numVarName;     // Name of the Igor complex numeric variable to set
double* realPtr;      // Pointer to real part of value to set it to
double* imagPtr;      // Pointer to imaginary part of value to set it to
int forceGlobal;      // Non-zero to create a global variable
```

Sets an Igor double-precision, complex numeric variable in the current data folder to a complex floating-point value after creating it, if necessary.

The function result is 0 or an error code.

If the named numeric variable already exists, SetIgorComplexVar just sets it. If it does not exist, it creates it and then sets it. In this case, it will create a global variable if forceGlobal is non-zero. If forceGlobal is zero, the variable will be local if a macro is running or global if not.

SetIgorComplexVar can not create a local variable in a user-defined function. Since it is usually undesirable for your XOP to behave differently depending on how it is called, you should generally avoid creating macro local variables.

In the event of a name conflict with a wave or string variable in the same scope, SetIgorComplexVar will return an error.

Prior to Igor Pro 5, the SetIgorComplexVar function was used to create output variables from external operations, analogous to V_flag. When programming for Igor Pro 5 or later, you should use Operation Handler (see page 151) to implement your external operation. Operation Handler will create output variables for you so you should not use SetIgorComplexVar for that purpose.

```
int
SetIgorStringVar(stringVarName, stringVarValue, forceGlobal)
char* stringVarName;        // Name of the Igor string variable to set
char* stringVarValue;       // Value to store in string variable
int forceGlobal;            // Non-zero to create a global string
```

Sets an Igor string variable in the current data folder after creating it, if necessary.

The function result is 0 or an error code.

If the named string variable already exists, SetIgorStringVar just sets it. If it does not exist, it creates it and then sets it. In this case, it will create a global string if forceGlobal is non-zero. If forceGlobal is zero, the string will be local if a macro is running or global if not.

SetIgorStringVar can not create a local variable in a user-defined function. Since it is usually undesirable for your XOP to behave differently depending on how it is called, you should generally avoid creating macro local variables.

In the event of a name conflict with a wave or numeric variable in the same scope, SetIgorStringVar will return an error.

Prior to Igor Pro 5, the SetIgorStringVar function was used to create output variables from external operations, analogous to S_fileName. When programming for Igor Pro 5 or later, you should use Operation Handler (see page 151) to implement your external operation. Operation Handler will create output variables for you so you should not use SetIgorStringVar for that purpose.

```
int
GetNVAR(nvp, realPartPtr, imagPartPtr, numTypePtr)
NVARRec* nvp;               // Pointer to an NVARRec field in a structure
double* realPartPtr;        // Real part of variable returned here
double* imagPartPtr;        // Imaginary part of variable returned here
int* numTypePtr;            // Numeric type of variable returned here
```

Retrieves the data and type of a global numeric variable referenced by an NVAR field in an Igor Pro structure.

nvp is a pointer to an NVAR field in an Igor structure. The Igor structure pointer would be passed into the XOP as a parameter.

If GetNVAR returns 0 then *realPartPtr will be the contents of the real part of the global variable and *imagPartPtr will be the contents of the imaginary part of the global variable, if it is complex.

realPartPtr and imagPartPtr must each point to storage for a double whether the global variable is complex or not.

*numTypePtr is set to the numeric type of the global. This will be either NT_FP64 or (NT_FP64 | NT_CMPLX).

Returns 0 or an error code.

Added for Igor Pro 5.03. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

See also **NVARs and SVARs In Structures** on page 283.

```
int
SetNVAR(nvp, realPartPtr, imagPartPtr)
NVARRec* nvp;               // Pointer to an NVARRec field in a structure
double* realPartPtr;        // Real part of variable supplied here
double* imagPartPtr;        // Imaginary part of variable supplied here
```

Sets the value of a global numeric variable referenced by an NVAR field in an Igor Pro structure.

nvp is a pointer to an NVAR field in an Igor structure. The Igor structure pointer would be passed into the XOP as a parameter.

*realPartPtr is the value to store in the real part of the global variable and *imagPartPtr is the value to store in the imaginary part of the global variable, if it is complex.

realPartPtr and imagPartPtr must each point to storage for a double whether the global variable is complex or not.

Returns 0 or an error code.

Added for Igor Pro 5.03. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

See also **NVARs and SVARs In Structures** on page 283.

```
int
GetSVAR(nvp, strHPtr)
NVARRec* nvp;             // Pointer to an NVARRec field in a structure
Handle* strHPtr;          // Handle for string variable returned here
```

Retrieves the handle for a global string variable referenced by an SVAR field in an Igor Pro structure.

svp is a pointer to an SVAR field in an Igor structure. The Igor structure pointer would be passed into the XOP as a parameter.

If GetSVAR returns 0 then *strHPtr will be the handle for an Igor global string variable.

**NOTE**: *strHPtr can be NULL if the global string variable contains no characters.

**NOTE**: *strHPtr belongs to Igor. Do not dispose it or alter it in any way. Use this function only to read the contents of the string.

Returns 0 or an error code.

Added for Igor Pro 5.03. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

See also **NVARs and SVARs In Structures** on page 283.

```
int
SetSVAR(nvp, strH)
NVARRec* nvp;             // Pointer to an NVARRec field in a structure
Handle* strH;             // Handle containing text for string variable
```

Sets the value of a global string variable referenced by an SVAR field in an Igor Pro structure.

svp is a pointer to an SVAR field in an Igor structure. The Igor structure pointer would be passed into the XOP as a parameter.

strH is a handle that you own. It can be NULL to set the global string variable to empty.

**NOTE**: Igor copies the contents of strH. You retain ownership of it and must dispose it.

Returns 0 or an error code.

Added for Igor Pro 5.03. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

See also **NVARs and SVARs In Structures** on page 283.

# Routines for Accessing Data Folders

Data folders, which provide hierarchical data storage, first appeared in Igor Pro 3.0. Most simple XOPs have no need to deal with them. For some advanced applications, being data-folder-aware will make your XOP more powerful. See **Dealing With Data Folders** on page 224 for an orientation to data folder use in XOPs.

Many of these routines require that you pass a data folder handle to Igor. You obtain a data folder handle from Igor using one of these routines:

```
GetDataFolderAndName     GetDataFolder              GetRootDataFolder
GetCurrentDataFolder     GetNamedDataFolder         GetDataFolderByIDNumber
GetParentDataFolder      GetIndexedChildDataFolder  GetDataFolderObject
GetWavesDataFolder       NewDataFolder
```

Data folder handles belong to Igor and your XOP must not delete or directly modify them.

The GetDataFolder and GetDataFolderAndName routines are used to implement the command parsing part of a data-folder-aware external operation. These are described under **Routines for Parsing Commands** on page 329.

```
int
GetDataFolderNameOrPath(dataFolderH, flags, dataFolderPathOrName)
DataFolderHandle dataFolderH;
int flags;
char dataFolderPathOrName[MAXCMDLEN+1];
```

Given a handle to a data folder, returns the name of the data folder if bit 0 of flags is zero or a full path to the data folder, including a trailing colon, if bit 0 of flags is set.

If bit 1 of flags is set, Igor returns the dataFolderPathOrName with single quotes if they would be needed to use the name or path in Igor's command line. If bit 1 of flags is zero, dataFolderPathOr-Name will have no quotes.

Set bit 1 of flags if you are going to use the path or name in a command that you submit to Igor via the XOPCommand or XOPSilentCommand callbacks. Clear bit 1 of flags for other purpose if you want an unquoted path or name.

All other bits of the flags parameter are reserved; you must set them to zero.

If dataFolderH is NULL, it uses the current data folder.

A data folder name can be up to MAX_OBJ_NAME characters while a full or partial path can be up to MAXCMDLEN characters. To be safe, allocate MAXCMDLEN+1 characters for dataFolderPath-OrName.

The function result is 0 or error code.

```
int
GetDataFolderIDNumber(dataFolderH, IDNumberPtr)
DataFolderHandle dataFolderH;
Long* IDNumberPtr;
```

Returns the unique ID number for the data folder via IDNumberPtr.

If dataFolderH is NULL, it uses the current data folder.

Each data folder has a unique ID number that stays the same as long as the data folder exists, even if it is renamed or moved. If you need to reference a data folder over a period of time during which it could be killed, then you should store the data folder's ID number.

Given the ID number, you can call GetDataFolderByIDNumber to check if the data folder still exists and to get a handle to it.

The ID number is valid until the user creates a new Igor experiment or quits Igor. ID numbers are not remembered from one running of Igor to the next.

The function result is 0 or error code.

```
int
GetDataFolderProperties(dataFolderH, propertiesPtr)
DataFolderHandle dataFolderH;
long* propertiesPtr;
```

Returns the bit-flag properties of the specified data folder.

If dataFolderH is NULL, it uses the current data folder.

At present, Igor does not support any properties and this routine will always return 0 in *propertiesPtr. In the future, it might support properties such as “locked”.

The function result is 0 or error code.

```
int
SetDataFolderProperties(dataFolderH, propertiesPtr)
DataFolderHandle dataFolderH;
long* propertiesPtr;
```

Sets the bit-flag properties of the specified data folder.

If dataFolderH is NULL, it uses the current data folder.

At present, Igor does not support any properties and this routine does nothing. In the future, it might support properties such as “locked”.

The function result is 0 or error code.

```
int
GetDataFolderListing(dataFolderH, optionsFlag, listingH)
DataFolderHandle dataFolderH;
int optionsFlag;      // Specifies what is to be listed
Handle listingH;      // Listing text is stored in handle.
```

Returns via listingH a listing of the contents of the specified data folder.

You must create listingH and dispose it when you no longer need it. Any contents in listingH are replaced by the listing.

Note that the text in the handle is *not* null terminated. Use GetHandleSize to find the number of bytes in the handle. To use C string functions on this text you need to copy it to a local buffer and null-terminate it or add a null terminator to the handle and lock the handle. See **Understand the Difference Between a String in a Handle and a C String** on page 321.

If dataFolderH is NULL, it uses the current data folder.

optionsFlag determines what is in the listing.

If bit 0 of optionsFlag is set, a list of subfolders is included:

```
"FOLDERS:<subFolder0>,<subFolder1>...,<subFolderN>;<CR>"
```

If bit 1 of optionsFlag is set, a list of waves is included:

```
"WAVES:<waveName0>,<waveName1>...,<waveNameN>;<CR>"
```

If bit 2 of optionsFlag is set, a list of numeric variables is included:

```
"VARIABLES:<varName0>,<varName1>...,<varNameN>;<CR>"
```

If bit 3 of optionsFlag is set, a list of string variables is included:

```
"STRINGS:<strVarName0>,<strVarName1>...,<strVarNameN>;<CR>"
```

All other bits are reserved and should be set to zero.

Names in the listing of waves, variables and strings are quoted with single quotes if this is necessary to make them suitable for use in the Igor command line.

The function result is 0 or error code.

***Example***

```
DataFolderHandle dataFolderH;
Handle listingH;
int result;

listingH = NewHandle(0L);
if (listingH == NULL)
   return NOMEM;
if (result = GetRootDataFolder(&dataFolderH)
   return result;
if (result = GetDataFolderListing(dataFolderH, 15, &listingH))
   return result;
<Use contents of listingH>
DisposeHandle(listingH);
```

```
int
GetRootDataFolder(refNum, rootDataFolderHPtr)
long refNum;                              // Not used - always pass zero.
DataFolderHandle* rootDataFolderHPtr;  // Root returned here.
```

Returns a handle to the root data folder in *rootDataFolderHPtr.

Data folder handles belong to Igor so you should not modify or dispose them.

The function result is 0 or error code.

```
int
GetCurrentDataFolder(currentDataFolderHPtr)
DataFolderHandle* currentDataFolderHPtr;  // Current returned here.
```

Returns a handle to the current data folder in *currentFolderHPtr.

Data folder handles belong to Igor so you should not modify or dispose them.

The function result is 0 or error code.

```
int
SetCurrentDataFolder(dataFolderH)
DataFolderHandle dataFolderH;
```

Sets the current data folder to the data folder referenced by dataFolderH.

The function result is 0 or error code.

```
int
GetNamedDataFolder(startingDataFolderH, dataFolderPath, dataFolderHPtr)
DataFolderHandle startingDataFolderH;
char dataFolderPath[MAXCMDLEN+1];
DataFolderHandle* dataFolderHPtr;
```

Returns in *dataFolderHPtr the data folder specified by startingDataFolderH and dataFolderPath.

Data folder handles belong to Igor so you should not modify or dispose them.

dataFolderPath can be an absolute path (e.g., "root:FolderA:FolderB:"), a relative path (e.g., ":FolderA:FolderB:") or a data folder name (e.g., "FolderA"). It can also be just ":", a form of relative path that refers to the starting data folder.

If dataFolderPath is an absolute path then startingDataFolderH is immaterial - you can pass any data folder handle or NULL. An absolute path must always start with "root:". It should include a trailing colon but GetNamedDataFolder tolerates an absolute path without the trailing colon. Note that "root" is an not an absolute path whereas "root:" is.

If dataFolderPath is a relative path or a data folder name then dataFolderPath is relative to startingDataFolderH. However, if startingDataFolderH is NULL then dataFolderPath is relative to the current folder.

Passing "root" as dataFolderPath will not find the root data folder because "root" is a data folder name. Igor will try to find a data folder named "root" relative to the current data folder. The actual root data folder is never relative to any data folder so it can not be found this way. Use "root:" instead.

Liberal names in dataFolderPath can be quoted or not. Both "root:Folder.1" and "root:'Folder.1'" are acceptable.

The function result is 0 or error code.

***Example***

```
DataFolderHandle rootH, dataFolderH;
int result;

if (result = GetRootDataFolderHandle(0, &rootH))
   return result;
if (result = GetNamedDataFolder(rootH, ":Packages:", &dataFolderH))
   return result;
```

```
int
GetDataFolderByIDNumber(IDNumber, dataFolderHPtr)
long IDNumber;
DataFolderHandle* dataFolderHPtr;
```

Returns via *dataFolderHPtr the data folder handle associated with the specified ID number.

Data folder handles belong to Igor so you should not modify or dispose them.

The function result is 0 if OK or a non-zero error code if the data folder doesn’t exist, which would be the case if the data folder were killed since you got its ID number.

Each data folder has a unique ID number that stays the same as long as the data folder exists. You can get the ID number for a data folder using GetDataFolderIDNumber.

If you need to reference a data folder over a period of time during which it could be killed, then you should store the data folder’s ID number. Given the ID number, GetDataFolderByIDNumber tells you if the data folder still exists and, if it does, gives you the data folder handle.

The ID number is valid until the user creates a new Igor experiment or quits Igor. ID numbers are not remembered from one running of Igor to the next.

```
int
GetParentDataFolder(dataFolderH, parentFolderHPtr)
DataFolderHandle dataFolderH;
DataFolderHandle* parentFolderHPtr;
```

Returns the parent of the specified data folder via *parentFolderHPtr.

Data folder handles belong to Igor so you should not modify or dispose them.

If dataFolderH is NULL, it uses the current data folder.

Passing the root data folder as dataFolderH is an error. In this case GetParentDataFolder returns NO_PARENT_DATAFOLDER.

The function result is 0 or error code.

```
int
GetNumChildDataFolders(parentDataFolderH, numChildDataFolderPtr)
DataFolderHandle parentFolderH;
long* numChildDataFolderPtr;
```

Returns the number of child data folders in the specified parent data folder.

If parentDataFolderH is NULL, it uses the current data folder.

The function result is 0 or error code.

```
int
GetIndexedChildDataFolder(parentDataFolderH, index,
childDataFolderHPtr)
DataFolderHandle parentFolderH;
long index;                              // 0-based index.
DataFolderHandle* childDataFolderHPtr;
```

Returns via childDataFolderHPtr a handle to the child data folder specified by the index.

Data folder handles belong to Igor so you should not modify or dispose them.

index starts from 0.

If parentDataFolderH is NULL, it uses the current data folder.

The function result is 0 or error code.

```
int
GetWavesDataFolder(wavH, dataFolderHPtr)
waveHndl wavH;
DataFolderHandle* dataFolderHPtr;
```

Returns via dataFolderHPtr the handle to the data folder containing the specified wave.

Data folder handles belong to Igor so you should not modify or dispose them.

The function result is 0 or error code.

```
int
NewDataFolder(parentFolderH, newDataFolderName, newDataFolderHPtr)
DataFolderHandle parentFolderH;
char newDataFolderName[MAX_OBJ_NAME+1];
DataFolderHandle* newDataFolderHPtr;
```

Creates a new data folder in the data folder specified by parentFolderH.

parentFolderH can be a handle to an Igor data folder or NULL to use the current data folder.

On output, *newDataFolderHPtr will contain a handle to the new data folder or NULL if an error occurred.

Data folder handles belong to Igor so you should not modify or dispose them.

NewDataFolder does not change the current data folder. If you want to make the new folder the current folder, call SetCurrentDataFolder after NewDataFolder.

The function result is 0 or error code.

```
int
KillDataFolder(dataFolderH)
DataFolderHandle dataFolderH;
```

Kills an existing data folder, removing it and its contents, including any child data folders, from memory.

dataFolderH is a handle to an existing Igor data folder or NULL to use the current data folder.

You will receive an error and the data folder will not be killed if it contains waves or variables that are in use (e.g., displayed in tables or graphs).

If you kill the current data folder or a data folder that contains the current data folder, Igor will set the current data folder to the parent of the killed data folder.

If you kill the root data folder, its contents will be killed but not the root data folder itself.

**NOTE**: Once a data folder is successfully killed, dataFolderH is no longer valid. You should not reference it for any purpose.

The function result is 0 or error code.

```
int
DuplicateDataFolder(sourceDataFolderH,parentDataFolderH,newDataFolderName)
DataFolderHandle sourceDataFolderH;
DataFolderHandle parentDataFolderH;
char newDataFolderName[MAX_OBJ_NAME+1];
```

Creates a clone of the source data folder. The contents of the destination will be clones of the contents of the source.

sourceDataFolderH is a handle to the data folder to be duplicated or NULL to use the current data folder.

parentDataFolderH is a handle to the data folder in which the new data folder is to be created or NULL to use the current data folder.

newDataFolderName is the name to be given to the new data folder.

The function result is 0 or error code.

```
int
MoveDataFolder(sourceDataFolderH, newParentDataFolderH)
DataFolderHandle sourceDataFolderH;
DataFolderHandle newParentDataFolderH;
```

Moves the source data folder into a new location in the hierarchy.

It is an error to attempt to move a parent folder into itself or one of its children.

sourceDataFolderH is a handle to the data folder to be moved or NULL to use the current data folder.

newParentDataFolderH is a handle to the data folder in which the source data folder is to be moved or NULL to use the current data folder.

The function result is 0 or error code.

```
int
RenameDataFolder(dataFolderH, newName)
DataFolderHandle dataFolderH;
char newName[MAX_OBJ_NAME+1];
```

Renames the data folder.

dataFolderH is a handle to the data folder to be renamed or NULL to use the current data folder.

The function result is 0 or error code.

```
int
GetNumDataFoldersObjects(dataFolderH, objectType, numObjectsPtr)
DataFolderHandle dataFolderH;
int objectType;
long* numObjectsPtr;
```

Returns via numObjectsPtr the number of objects of the specified type in the specified data folder.

If dataFolderH is NULL, it uses the current data folder.

objectType is one of the following:

| | |
|---|---|
| WAVE_OBJECT | for waves |
| VAR_OBJECT | for numeric variables |
| STR_OBJECT | for string variables |

To find the number of data folders within the data folder, use GetNumChildDataFolders.

```
int
GetIndexedDataFolderObject(dataFolderH,objectType,index,objectName,vp)
DataFolderHandle dataFolderH;
int objectType;
long index;
char objectName[MAX_OBJ_NAME+1];
DataObjectValuePtr vp;
```

Returns information that allows you to access an object of the specified type in the specified data folder.

index starts from 0.

If dataFolderH is NULL, it uses the current data folder.

objectType is one of the following:

| | |
|---|---|
| WAVE_OBJECT | for waves |
| VAR_OBJECT | for numeric variables |
| STR_OBJECT | for string variables |

For information on a data folder, use GetIndexedChildDataFolder.

You can pass NULL for objectName if you don’t need to know the name of the object.

If you do not want to get the value of the object, pass NULL for vp.

If vp is not NULL, then it is a pointer to a DataObjectValue union, defined in IgorXOP.h. GetIndexedDataFolderObject sets fields depending on the object’s type:

| | |
|---|---|
| WAVE_OBJECT | Sets vp->wavH field to wave's handle. |
| VAR_OBJECT | Stores numeric variable's value in vp->nv field. |
| STR_OBJECT | Sets vp->strH field to strings's handle. |

The handles returned via the wavH and strH fields belong to Igor. Do not modify or dispose them.

The function result is 0 or error code.

```
int
GetDataFolderObject(dataFolderH, objectName, objectTypePtr, vp)
DataFolderHandle dataFolderH;
char objectName[MAX_OBJ_NAME+1];
int* objectTypePtr;
DataObjectValuePtr vp;
```

Returns information about the named object in the specified data folder.

The main utility of this routine will be for getting access to a numeric or string variable in a specific data folder.

If dataFolderH is NULL, it uses the current data folder.

On output, if the specified object exists, *objectTypePtr will be one of the following:

| | |
|---|---|
| WAVE_OBJECT | Object is a wave. |
| VAR_OBJECT | Object is a numeric variable. |
| STR_OBJECT | Object is a string variable. |
| DATAFOLDER_OBJECT | Object is a data folder. |

If you do not want to get the value of the object, pass NULL for vp.

If vp is not NULL, then it is a pointer to a DataObjectValue union, defined in IgorXOP.h. GetIndexedDataFolderObject sets fields depending on the object's type:

| | |
|---|---|
| WAVE_OBJECT | Sets vp->wavH field to wave's handle. |
| VAR_OBJECT | Stores numeric variable's value in vp->nv field. |
| STR_OBJECT | Sets vp->strH field to strings's handle. |
| DATAFOLDER_OBJECT | Sets vp->dfH field to data folder's handle. |

The handles returned via the wavH, strH and dfH fields belong to Igor. Do not modify or dispose them. Remember also that strings in handles do not contain a null terminator (they are not C strings). To find the number of characters, call GetHandleSize on the handle.

The function result is 0 or error code if there is no object with the specified name.

*Example*

```
DataObjectValue v;
int objectType;
waveHndl wavH;
Handle strH;
double numVarValue;
DataFolderHandle dfH;
int err;

if (err = GetDataFolderObject(dfH, "AnObject", &objectType, &v))
   return err;
switch(objectType) {
   case WAVE_OBJECT:
      wavH = v.wavH;
      <Do something with wavH>;
      break;
   case VAR_OBJECT:
      numVarValue = v.nv.realValue;
      <Do something with numVarValue>;
      break;
   case STR_OBJECT:
      strH = v.strH;
      <Do something with strH>;
      break;
   case DATAFOLDER_OBJECT:
      dfH = v.dfH;
      <Do something with dfH>;
      break;
}
```

```
int
SetDataFolderObject(dataFolderH, objectName, objectType, vp)
DataFolderHandle dataFolderH;
char objectName[MAX_OBJ_NAME+1];
int objectType;
DataObjectValuePtr vp;
```

Sets the value of the named object in the specified data folder.

The main utility of this routine will be for setting the value of a numeric or string variable in a specific data folder.

If dataFolderH is NULL, it uses the current data folder.

objectType must be one of the following:

| | |
|---|---|
| WAVE_OBJECT | Object is a wave. |
| VAR_OBJECT | Object is a numeric variable. |
| STR_OBJECT | Object is a string variable. |
| DATAFOLDER_OBJECT | Object is a data folder. |

vp is a pointer to a DataObjectValue union, defined in IgorXOP.h. The action of SetIndexedDataFolderObject depends on the object's type:

| | |
|---|---|
| WAVE_OBJECT | Does nothing. |
| VAR_OBJECT | Sets numeric variable to the value in vp->nv field. |
| STR_OBJECT | Sets the string variable to contain the characters in vp->strH. |
| DATAFOLDER_OBJECT | Does nothing. |

When setting the value of a string variable, Igor just copies the data from the handle. The handle is yours to dispose (if you created it). Remember also that strings in handles do not contain a null terminator (they are not C strings). Remove the null terminator if you have added one before calling SetDataFolderObject.

The function result is 0 or error code if there is no object with the specified name.

***Example***

```
DataObjectValue v;
int objectType;
waveHndl wavH;
Handle strH;
double numVarValue;
DataFolderHandle dfH;
int err;

if (err = GetDataFolderObject(dfH, "AnObject", &objectType, &v))
   return err;
switch(objectType) {
   case WAVE_OBJECT:
      break;
   case VAR_OBJECT:
      v.realPart += 1;                 // Increment the numeric variable
      break;
   case STR_OBJECT:
      strH = v.strH;                   // This handle belongs to Igor.
      if (HandToHand(&strH))           // Make our own copy.
         return NOMEM;
      if (PtrAndHand("***", strH, 3)) {   // Append *** to string
         DisposeHandle(strH);
         return NOMEM;
      }
      v.strH = strH;                   // Used by SetDataFolderObject.
      break;
   case DATAFOLDER_OBJECT:
      break;
}
// Note: SetDataFolderObject does nothing for waves or data folders.
err = SetDataFolderObject(dfH, "AnObject", objectType, &v);
if (objectType==STR_OBJECT)
   DisposeHandle(strH);                // Dispose our copy of handle.
return err;
```

```
int
KillDataFolderObject(dataFolderH, objectType, objectName)
DataFolderHandle dataFolderH;
int objectType;
char objectName[MAX_OBJ_NAME+1];
```

Kills the named object of the specified type in the specified data folder.

If dataFolderH is NULL, it uses the current data folder.

objectType is one of the following:

| | |
|---|---|
| WAVE_OBJECT | for waves |
| VAR_OBJECT | for numeric variables |
| STR_OBJECT | for string variables |

**NOTE**: If you attempt to kill a wave that is in use (e.g., in a graph, table or user-defined function) the wave will not be killed and you will receive a non-zero result code.

Igor does not check if numeric and string variables are in use. You can kill a numeric or string variable at any time without receiving an error.

The function result is 0 or error code.

```
int
DuplicateDataFolderObject(dataFolderH, objectType, objectName,
                              destFolderH,   newObjectName, overwrite)
DataFolderHandle dataFolderH;
int objectType;
char objectName[MAX_OBJ_NAME+1];
DataFolderHandle destFolderH;
char newObjectName[MAX_OBJ_NAME+1];
int overwrite;
```

Duplicates the named object of the specified type.

If dataFolderH and/or destFolderH is NULL, it uses the current data folder.

objectType is one of the following:

| | |
|---|---|
| WAVE_OBJECT | for waves |
| VAR_OBJECT | for numeric variables |
| STR_OBJECT | for string variables |

If the new name is illegal you will receive a non-zero result code.

If the new name is in use and overwrite is false, you will receive a non-zero result code.

If the new name is in use for a different kind of object, you will receive a non-zero result code.

To avoid these errors, you can check and if necessary fix the new name using the CheckName, CleanupName and UniqueName2 routines or the higher-level CreateValidDataObjectName routine.

The function result is 0 or error code.

```
int
MoveDataFolderObject(sourceDataFolderH, objectType, objectName,
                                                     destDataFolderH)
DataFolderHandle sourceDataFolderH;
int objectType;
char objectName[MAX_OBJ_NAME+1];
DataFolderHandle destDataFolderH;
```

Moves the named object of the specified type from the source data folder to the destination data folder.

If sourceDataFolderH is NULL, it uses the current data folder.

If destDataFolderH is NULL, it uses the current data folder.

objectType is one of the following:

| | |
|---|---|
| WAVE_OBJECT | for waves |
| VAR_OBJECT | for numeric variables |
| STR_OBJECT | for string variables |

**NOTE**: If an object with the same name exists in the destination data folder, the object will not be moved and you will receive a non-zero result code.

The function result is 0 or error code.

```
int
RenameDataFolderObject(dataFolderH, objectType, objectName,
                                                        newObjectName)
DataFolderHandle dataFolderH;
int objectType;
char objectName[MAX_OBJ_NAME+1];
char newObjectName[MAX_OBJ_NAME+1];
```

Renames the named object of the specified type in the specified data folder.

If dataFolderH is NULL, it uses the current data folder.

objectType is one of the following (defined in IgorXOP.h):

| | |
|---|---|
| WAVE_OBJECT | for waves |
| VAR_OBJECT | for numeric variables |
| STR_OBJECT | for string variables |

**NOTE**: If the new name is illegal or in use the object will not be renamed and you will receive a non-zero result code.

The function result is 0 or error code.

# Routines for XOPs with Menu Items

If your XOP adds one or more menu items to Igor then it must enable and disable them or change them according to circumstances. It must also respond properly when its menu item is selected. These routines allow you to manage your menu items.

See Chapter 8, **Adding Menus and Menu Items** for an overview.

In dealing with menus, XOPs use Macintosh Menu Manager routines, even when running on Windows. See **Menu Manager Routines** on page 232 for a list of these routines.

Routines for dealing with dialog popup menus are described under **Dialog Popup Menus** on page 405.

```
int
SetIgorMenuItem(message, enable, text, param)
int message;                // An Igor message code
int enable;                 // 1 to enable the menu item, 0 to disable it
char* text;                 // Pointer to a C string or NULL
long param;                 // Normally not used and should be 0
```

Enables or disables the built-in Igor menu item associated with message.

For example, if the XOP wants to enable the Copy menu item, it would call Igor as follows:

```
SetIgorMenuItem(COPY, 1, NULL, 0);
```

COPY is the event message code that would be sent by Igor if the user chose the Copy item in the Edit menu. Event message codes are defined in XOP.h.

The text parameter will normally be NULL. However, there are certain built-in Igor menus whose text can change. An example of this is the Undo item. An XOP which owns the active window can set the Undo item as follows:

```
SetIgorMenuItem(UNDO, 1, "Undo XOP-Specific Action", 0);
```

Igor will ignore the text parameter for menu items whose text is fixed, for example Copy. For menu items whose text is variable, if the text parameter is not NULL, then Igor will set the text of the menu item to the specified text.

The param parameter is normally not used and should be zero. There is currently only one case in which it is used. If the message is FIND, Igor needs to know if you want Find, Find Same or Find Selected Text. It looks at the param parameter for this which should be 1, 2 or 3, respectively.

Returns 1 if there is a menu item corresponding to message or 0 if not. Normally you will have no need for this return value.

```
int
ResourceToActualMenuID(resourceMenuID)
int resourceMenuID;
```

Given the ID of a MENU resource in the XOP's resource fork, returns the actual menu ID of that resource in memory.

Returns 0 if the XOP did not add this menu to Igor.

See **Determining Which Menu Item Was Chosen** on page 241 for a discussion of resource IDs versus actual IDs.

```
int
ActualToResourceMenuID(menuID)
int menuID;
```

Given the ID of a menu in memory, returns the resource ID of the MENU resource in the XOP's resource fork.

Returns 0 if the XOP did not add this menu to Igor.

See **Determining Which Menu Item Was Chosen** on page 241 for a discussion of resource IDs versus actual IDs.

```
int
ResourceToActualItem(igorMenuID, resourceItemNumber)
int igorMenuID;
int resourceItemNumber;
```

Given the ID of a built-in Igor menu and the number of a menu item specification in the XMI1 resource in the XOP's resource fork, returns the actual item number of that item in the Igor menu.

Both menu item specification numbers and menu item numbers start from one.

Returns 0 if the XOP did not add this menu item to Igor.

See **Determining Which Menu Item Was Chosen** on page 241 for a discussion of resource items versus actual items.

```
int
ActualToResourceItem(igorMenuID, actualItemNumber)
int igorMenuID;
int actualItemNumber;
```

Given the ID of a built-in Igor menu and the actual number of a menu item in the Igor menu, returns the number of the menu item specification in the XMI1 resource in the XOP's resource fork for that item.

Both menu item specification numbers and menu item numbers start from one.

Returns 0 if the XOP did not add this menu item to Igor.

See **Determining Which Menu Item Was Chosen** on page 241 for a discussion of resource items versus actual items.

```
MenuHandle
ResourceMenuIDToMenuHandle(resourceMenuID)
int resourceMenuID;
```

Given the ID of a MENU resource in the XOP's resource fork, returns the menu handle for that menu.

Returns NULL if XOP did not add this menu.

```
void
WMDeleteMenuItems(theMenu, afterItem)
MenuHandle theMenu;  // Handle to a popup menu
int afterItem;       // The number of an item in the menu
```

Deletes all of the items after the specified item in the menu.

Item numbers start from one.

Do not call this routine to update the contents of a dialog popup menu item. Use DeletePopMenuItems instead.

Prior to Carbon this was called DeleteMenuItems. In the Carbon API, Apple stole that name.

```
void
FillMenu(theMenu, itemList, itemListLen, afterItem)
MenuHandle theMenu;  // Handle to a popup menu
char* itemList;      // Semicolon separated list of items to add to menu
long itemListLen;    // Number of characters in the itemList
int afterItem;       // The number of an item in the menu
```

Puts the items in itemList into theMenu after the specified item.

If afterItem is 0, the new items go at the beginning of the menu.

This routine supports Macintosh menu manager meta-characters in menu items. For example, if a "(" character appears in the item list, it will not be displayed in the corresponding menu item but instead will cause the item to be disabled.

Do not call this routine to update the contents of a dialog popup menu item. Use FillPopMenu instead.

```
void
FillMenuNoMeta(theMenu, itemList, itemListLen, afterItem)
MenuHandle theMenu;  // Handle to a popup menu
char* itemList;      // Semicolon separated list of items to add to menu
long itemListLen;    // Number of characters in the itemList
int afterItem;       // The number of an item in the menu
```

Puts the items in itemList into theMenu after the specified item.

If afterItem is 0, the new items go at the beginning of the menu.

Unlike FillMenu, this routine does not support meta-characters in menu items. For example, if a "(" character appears in the item list, it will be displayed in the corresponding menu item rather than being interpreted as disabling the item.

Do not call this routine to update the contents of a dialog popup menu item. Use FillPopMenu instead.

```
int
FillWaveMenu(theMenu, match, options, afterItem)
MenuHandle theMenu;  // Handle to a popup menu
char* match;         // "*" for all waves or match pattern
char* options;       // Options for further selection of wave
int afterItem;       // The number of an item in the menu
```

Fills the menu with wave names selected based on match and options.

The added items are added after the item specified by afterItem or, if afterItem is 0, at the beginning of the menu.

Returns zero if everything went OK or an error code if the parameters were not legal or another problem (such as out of memory) occurred.

The meaning of the match, and options parameters is the same as for the built-in Igor WaveList function.

Do not call this routine to update the contents of a dialog popup menu item. Use FillWavePopMenu instead.

In contrast to Macintosh menu manager routines, this routine does not treat any characters as meta-characters.

```
int
FillPathMenu(theMenu, match, options, afterItem)
MenuHandle theMenu;  // Handle to a popup menu
char* match;         // "*" for all paths or match pattern
char* options;       // Options for further selection of path
int afterItem;       // The number of an item in the menu
```

Fills the menu with path names selected based on match.

The added items are added after the item specified by afterItem or, if afterItem is 0, at the beginning of the menu.

Returns zero if everything went OK or an error code if the parameters were not legal or another problem (such as out of memory) occurred.

The meaning of the match parameter is the same as for the built-in Igor PathList function.options must be "".

Do not call this routine to update the contents of a dialog popup menu item. Use FillPathPopMenu instead.

In contrast to Macintosh menu manager routines, this routine does not treat any characters as meta-characters.

```
int
FillWinMenu(theMenu, match, options, afterItem)
MenuHandle theMenu;  // Handle to a popup menu
char* match;         // "*" for all windows or match pattern
char* options;       // Options for further selection of windows
int afterItem;       // The number of an item in the menu
```

Fills the menu with Igor target window names selected based on match and options.

The added items are added after the item specified by afterItem or, if afterItem is 0, at the beginning of the menu.

Returns zero if everything went OK or an error code if the parameters were not legal or another problem (such as out of memory) occurred.

The meaning of the match parameter is the same as for the built-in Igor WinList function.

If options is "" then all windows are selected.

If options is "WIN:" then just the target window is selected.

If options is "WIN:typeMask" then windows of the specified types are selected.

The window type masks are defined in IgorXOP.h. For example, "WIN:1" selects graph windows only. "WIN:3" selects graphs and tables.

Do not call this routine to update the contents of a dialog popup menu item. Use FillWindowPopMenu instead.

In contrast to Macintosh menu manager routines, this routine does not treat any characters as meta-characters.

# Routines for XOPs that Have Dialogs

These utilities assist an XOP in handling a standard modal dialog. See the sample file VDTDialog.c for an example of a modal dialog. See GBLoadWaveDialog.c for an example of an Igor-style modal dialog. See **Adding Dialogs** on page 269 for an overview.

```
DialogPtr
GetXOPDialog(dialogID)
int dialogID;
```

This routine is supported on Macintosh only. There is no Windows equivalent.

This utility routine works just like the Macintosh GetNewDialog toolbox routine. XOPs must call GetXOPDialog instead of GetNewDialog.

```
void
ShowDialogWindow(theDialog)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
```

Shows the dialog window if it is hidden.

Use this in place of the Mac OS ShowWindow call for platform independence.

```
CGrafPtr or XOP_DIALOG_REF
SetDialogPort(theDialog)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
```

Sets the current GrafPort on Macintosh and does nothing on Windows.

On Macintosh, SetDialogPort returns the current GrafPort before it was called. On Windows, it returns theDialog. This routine exists solely to avoid the need for an ifdef when you need to deal with the Macintosh current GrafPort in cross-platform dialog code.

```
void
DoXOPDialog(itemPtr)
short* itemPtr;
```

This routine is supported on Macintosh only. There is no Windows equivalent.

DoXOPDialog works like the Macintosh ModalDialog toolbox routine except that it does not take a dialog filter procedure as a parameter. Instead, it uses a filter routine that converts a return key or enter key press into a click on the default dialog item (usually item number 1). Macintosh XOPs must call XOPDialog (described below) or DoXOPDialog instead of calling ModalDialog directly.

If your dialog requires a more complex filter routine, then you will have to call XOPDialog on the Macintosh, passing it your own filter routine.

See GBLoadWaveDialog.c for an example.

```
void
XOPDialog(filterProc, itemPtr)
ModalFilterUPP filterProc;
short* itemPtr;
```

This routine is supported on Macintosh only. There is no Windows equivalent.

XOPDialog works just like the Macintosh ModalDialog toolbox routine. Macintosh XOPs must call XOPDialog or DoXOPDialog (described above) instead of calling ModalDialog directly.

```
void
DisposeXOPDialog(theDialog)
DialogPtr theDialog;
```

This routine is supported on Macintosh only. There is no Windows equivalent.

This utility routine works just like the Macintosh DisposeDialog toolbox routine. Macintosh XOPs must call DisposeXOPDialog instead of DisposeDialog.

```
void
SetDialogBalloonHelpID(balloonHelpID)
int balloonHelpID;   // ID of the balloon help resource for the dialog
```

On the Macintosh, sets the resource ID for the hdlg resource to be used for balloon help. If balloonHelpID is -1, this indicates that no balloon help is to be used.

On Windows, this routine does nothing.

This routine is of limited use because balloon help is not supported on Mac OS X. See **Macintosh Balloon Help** on page 295 for details.

```
void
GetDBox(theDialog, itemNumber, box)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
Rect* box;                 // Pointer to rect to receive item's rectangle
```

Given a dialog and the item number of an item in that dialog, returns the item’s rectangle via box.

On Macintosh, the returned rectangle is in the local coordinates of the dialog window. On Windows, the returned rectangle is in client window coordinates.

```
int
GetRadBut(theDialog, itemNumber)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
```

Given a dialog and the item number of a radio button in the dialog, returns zero if the radio button is not turned on, non-zero if it is turned on.

```
void
SetRadBut(theDialog, first, last, theButton)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int first, last, theButton;
```

Given a dialog, a range of item numbers of a group of radio buttons, and the item number for the radio button which should be on, turns that radio button on and others in the group off.

```
int
ToggleCheckBox(theDialog, itemNumber)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
```

Given a dialog and the item number of a checkbox in that dialog, toggles the state of the checkbox. Returns the new state (0 = off, 1 = on).

```
int
GetCheckBox(theDialog, itemNumber)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
```

Given a dialog and the item number of a checkbox in that dialog, returns the state of the checkbox (0 = off, 1 = on).

```
void
SetCheckBox(theDialog, itemNumber, val)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
int val;
```

Given a dialog, the item number of a checkbox in that dialog, and a value to set the checkbox to (0 = off, 1 = on), sets the state of the checkbox.

```
void
HiliteDControl(theDialog, itemNumber, enable)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
int enable;
```

Given a dialog and the item number of a control item in that dialog, enables or disables the control and sets its highlighting to reflect its state.

The control is enabled if enable is non-zero, disabled and grayed out otherwise.

```
void
EnableDControl(theDialog, itemNumber)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
```

Given a dialog and the item number of a control item in that dialog, enables the control and sets its highlighting to reflect its enabled state.

```
void
DisableDControl(theDialog, itemNumber)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
```

Given a dialog and the item number of a control item in that dialog, disables the control and sets its highlighting to reflect its disabled state (grays the control out).

```
int
GetDText(theDialog, itemNumber, theText)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
char* theText;             // C string to contain up to 255 chars from
item
```

Given a dialog and the item number of an edit text or static text item in that dialog, returns the text in the item via theText.

The string returned is a C string of up to 255 characters. theText should be big enough for 255 characters plus the null terminator.

The function result is the number of characters in the string returned via theText.

```
void
SetDText(theDialog, itemNumber, theText)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
char* theText;             // C string containing text for dialog item
```

Given a dialog and the item number of an edit text or static text item in that dialog, sets the text in the item to the contents of theText.

```
int
GetDInt(theDialog, itemNumber, theInt)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
int* theInt;               // Pointer to int to receive number in item
```

Given a dialog and the item number of an edit text item in that dialog, returns the number entered in the item via theInt.

The function result is zero if a number was read from the item or non-zero if no number could be read because the item had no text in it or the text was not a valid number.

```
void
SetDInt(theDialog, itemNumber, theInt)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
int theInt;                // The number to put in edit text item
```

Given a dialog and the item number of an edit text item in that dialog, sets the text in the item to the number in theInt.

```
int
GetDLong(theDialog, itemNumber, theLong)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
long* theLong;             // Pointer to long to receive number in item
```

Given a dialog and the item number of an edit text item in that dialog, returns the number entered in the item via theLong.

The function result is zero if a number was read from the item or non-zero if no number could be read because the item had no text in it or the text was not a valid number.

```
void
SetDLong(theDialog, itemNumber, theLong)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
long theLong;              // The number to put in edit text item
```

Given a dialog and the item number of an edit text item in that dialog, sets the text in the item to the number in theLong.

```
int
GetDDouble(theDialog, itemNumber, theDouble)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
double* theDouble;         // Pointer to double to receive number in item
```

Given a dialog and the item number of an edit text item in that dialog, returns the number entered in the item via theDouble.

The function result is zero if a number was read from the item or non-zero if no number could be read because the item had no text in it or the text was not a valid number.

```
void
SetDDouble(theDialog, itemNumber, theDouble)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
double* theDouble;         // Pointer to double to put in edit text item
```

Given a dialog and the item number of an edit text item in that dialog, sets the text in the item to the number pointed to by theDouble.

```
void
SelEditItem(theDialog, itemNumber)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
```

Selects the entire text of the edit text item specified by itemNumber.

itemNumber must be the dialog item number of an editText item. Prior to Carbon, if itemNumber were 0, SelEditItem selected the text in the current edit text item. This is no longer supported.

If the dialog has no edit items, it does nothing.

This routine is used to preselect an entire edit item so that the user does not have to select text before starting to type. This behavior is desirable on Macintosh but usually is not desirable on Windows. It is usually appropriate to call SelMacEditItem instead of SelEditItem.

```
void
SelMacEditItem(theDialog, itemNumber)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int itemNumber;
```

Selects the entire text of the edit text item specified by itemNumber.

itemNumber must be the dialog item number of an editText item. Prior to Carbon, if itemNumber were 0, SelMacEditItem selected the text in the current edit text item. This is no longer supported.

If the dialog has no edit items, it does nothing.

SelMacEditItem is the same as SelEditItem except that it does nothing on Windows.

This routine is used to preselect an entire edit item so that the user does not have to select text before starting to type. This behavior is desirable on Macintosh but usually is not desirable on Windows. The difference stems from the fact that on Macintosh an edit text item almost always has the focus whereas on Windows, any item can have the focus. It is recommended to call SelMacEditItem(theDialog, 0) just before you enter the main dialog loop.

```
void
DisplayDialogCmd(theDialog, dlogItemNo, cmd)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int dlogItemNo;            // Dialog item in which cmd is displayed
char* cmd;                 // The command to be displayed
```

Displays the command in an Igor-style dialog that has Do It, To Cmd, and To Clip buttons. See GBLoadWaveDialog.c for an example.

dlogItemNo is the item number of the dialog item in which the command is to be displayed. On the Macintosh, this must be a user item. On Windows, it must be an EDITTEXT item.

```
void
FinishDialogCmd(cmd, mode)
char* cmd;
int mode;
```

You call FinishDialogCmd at end of an Igor-style dialog.

cmd is a C string containing the command generated by the Igor-style dialog.

If mode is 1, FinishDialogCmd puts the command in Igor’s command line and starts execution.

If mode is 2, FinishDialogCmd puts the command in Igor’s command line but does not start execution.

If mode is 3, FinishDialogCmd puts the command in the clipboard.

Liberal names of waves and data folders must be quoted before using them in the Igor command line. Use PossiblyQuoteName when preparing the command to be executed so that your XOP works with liberal names.

## Dialog Popup Menus

These routines provide a platform-independent way to implement popup menus in dialogs. For an overview of dialog popup menu support, see **Cross-Platform Dialog Popup Menus** on page 275.

```
void
InitPopMenus(theDialog)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
```

Call this during initialization of a dialog that uses popup menus before calling any other popup menu related routines.

If you call InitPopMenus you must also call KillPopMenus when the dialog is finished.

```
void
CreatePopMenu(theDialog,popupItemNum,titleItemNum,itemList,initialItem)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int popupItemNum;          // Dialog item number for popup menu
int titleItemNum;          // Dialog item number for popup title
const char* itemList;      // List of initial popup menu items
int initialItem;           // Number of initially-selected item
```

Creates a dialog popup menu using a popup menu on Macintosh and a drop-down list combo box on Windows. We use the term "popup menu" to mean a popup menu in a Macintosh dialog or a combo box in a Windows dialog.

popupItemNum is the dialog item number for the popup menu. On Macintosh, this must be specified as a control in the DITL resource and there must be a corresponding CNTL resource. See GBLoadWave.r for an example. On Windows, it must be a COMBOBOX with the style (CBS_DROPDOWNLIST | CBS_HASSTRINGS). See GBLoadWave.rc for an example.

titleItemNum is the dialog item number for the static text title for the popup menu. Prior to Carbon, on Macintosh this item was highlighted when the user clicked on the popup menu. As of the Carbon, it is no longer used but must be present for backward compatibility.

itemList is a semicolon-separated list of items to insert into the menu. For example, "Red;Green;Blue;".

initialItem is the 1-based number of the item in the popup menu that should be initially selected.

In contrast to Macintosh menu manager routines, this routine does not treat any characters as meta-characters.

```
MenuHandle
GetPopMenuHandle(theDialog, dlogItemNo)
XOP_DIALOG_REF theDialog;  // DialogPtr on Macintosh, HWND on Windows.
int dlogItemNo;            // Dialog item number.
```

This routine is supported on Macintosh only. There is no Windows equivalent.

Returns the menu handle for the specified dialog popup menu or NULL if the specified item is not an initialized dialog popup menu.

Using the menu handle returned by GetPopMenuHandle, you can enable and disable items. However, this works on Macintosh only. For cross-platform XOPs, other methods must be found. For example, instead of disabling inappropriate items, you can remove them from the popup menu or display an error message if they are selected.

```
int
ItemIsPopMenu(theDialog, dlogItemNo)
XOP_DIALOG_REF theDialog;  // DialogPtr on Macintosh, HWND on Windows.
int dlogItemNo;            // Dialog item number.
```

Returns the truth that the item is a popup menu. On Windows it returns true if the item is a combo box.

This routine is used by the HandleItemHit routine in the GBLoadWaveDialog.c and VDTDialog.c files. It allows HandleItemHit to respond to popup menu items in a different manner than other kinds of items. This is the only intended use for this routine.

```
int
AddPopMenuItems(theDialog, dlogItemNo, itemList)
XOP_DIALOG_REF theDialog;  // DialogPtr on Macintosh, HWND on Windows.
int dlogItemNo;            // Dialog item number.
const char* itemList;      // List of items to be added.
```

Adds the contents of itemList to the existing dialog popup menu.

This can be called only after the popup menu item has been initialized by calling CreatePopMenu.

itemList can be a single item ("Red") or a semicolon-separated list of items ("Red;Green;Blue;"). The trailing semicolon is optional.

In contrast to Macintosh menu manager routines, this routine does not treat any characters as meta-characters.

```
int
SetPopMatch(theDialog, dlogItemNo, selStr)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int dlogItemNo;            // Item number of the popup menu item
char* selStr;              // Text of the item to be selected
```

Selects the item in the popup that matches selStr.

The match is case insensitive.

Returns the number of the menu item selected or zero if there is no match.

```
void
SetPopItem(theDialog, dlogItemNo, theItem)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int dlogItemNo;            // Item number of the popup menu item
int theItem;               // Number of the menu item to select
```

Makes the item the currently selected item in the popup.

Item numbers in menus start from one.

```
void
GetPopMenu(theDialog, dlogItemNo, selItem, selStr)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int dlogItemNo;            // Item number of the popup menu item
int* selItem;              // Receives selected item number
char* selStr;              // Receives text of item selected
```

Returns the item number and text of the currently selected item in the popup.

This can be called only after the popup menu item has been initialized by calling CreatePopMenu.

If you are not interested in the text of the item selected, pass NULL for selStr.

```
void
DeletePopMenuItems(theDialog, dlogItemNo, afterItem)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int dlogItemNo;            // Dialog item number for popup menu
int afterItem;             // The number of an item in the menu
```

Deletes all of the items after the specified item in the dialog popup menu.

This can be called only after the popup menu item has been initialized by calling CreatePopMenu.

Item numbers start from one. Pass 0 to delete all items.

```
void
FillPopMenu(theDialog, dlogItemNo, itemList, itemListLen, afterItem)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int dlogItemNo;            // Dialog item number for popup menu
char* itemList;            // Semicolon separated list of items to add
long itemListLen;          // Number of characters in the itemList
int afterItem;             // The number of an item in the menu
```

Sets the contents of the existing dialog popup menu.

This can be called only after the popup menu item has been initialized by calling CreatePopMenu.

afterItem is 1-based. The added items are added after the item specified by afterItem or, if afterItem is 0, at the beginning of the popup menu.

In contrast to Macintosh menu manager routines, this routine does not treat any characters as meta-characters.

```
int
FillWavePopMenu(theDialog, dlogItemNo, match, options, afterItem)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int dlogItemNo;            // Dialog item number for popup menu
char* match;               // "*" for all waves or match pattern
char* options;             // Options for further selection of wave
int afterItem;             // The number of an item in the menu
```

Fills the dialog popup menu with wave names selected based on match and options.

This can be called only after the popup menu item has been initialized by calling CreatePopMenu.

afterItem is 1-based. The added items are added after the item specified by afterItem or, if afterItem is 0, at the beginning of the popup menu.

Returns zero if everything went OK or an error code if the parameters were not legal or another problem (such as out of memory) occurred.

The meaning of the match, and options parameters is the same as for the Igor WaveList function.

In contrast to Macintosh menu manager routines, this routine does not treat any characters as meta-characters.

```
int
FillPathPopMenu(theDialog, dlogItemNo, match, options, afterItem)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int dlogItemNo;            // Dialog item number for popup menu
char* match;               // "*" for all paths or match pattern
char* options;             // Options for further selection of path
int afterItem;             // The number of an item in the menu
```

Fills the dialog popup menu with path names selected based on match.

This can be called only after the popup menu item has been initialized by calling CreatePopMenu.

afterItem is 1-based. The added items are added after the item specified by afterItem or, if afterItem is 0, at the beginning of the popup menu.

Returns zero if everything went OK or an error code if the parameters were not legal or another problem (such as out of memory) occurred.

The meaning of the match parameter is the same as for the built-in Igor PathList function.options must be "".

In contrast to Macintosh menu manager routines, this routine does not treat any characters as meta-characters.

```
int
FillWindowPopMenu(theDialog, dlogItemNo, match, options, afterItem)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
int dlogItemNo;            // Dialog item number for popup menu
char* match;               // "*" for all windows or match pattern
char* options;             // Options for further selection of windows
int afterItem;             // The number of an item in the menu
```

Fills the dialog popup menu with Igor target window names selected based on match and options.

This can be called only after the popup menu item has been initialized by calling CreatePopMenu.

afterItem is 1-based. The added items are added after the item specified by afterItem or, if afterItem is 0, at the beginning of the popup menu.

Returns zero if everything went OK or an error code if the parameters were not legal or another problem (such as out of memory) occurred.

The meaning of the match parameter is the same as for the built-in Igor WinList function.

If options is "" then all windows are selected.

If options is "WIN:" then just the target window is selected.

If options is "WIN:typeMask" then windows of the specified types are selected.

The window type masks are defined in IgorXOP.h. For example, "WIN:1" selects graph windows only. "WIN:3" selects graphs and tables.

In contrast to Macintosh menu manager routines, this routine does not treat any characters as meta-characters.

```
void
KillPopMenus(theDialog)
XOP_DIALOG_REF theDialog;  // DialogPtr on Mac, HWND on Windows
```

Cleans up and disposes popup menu handles.

Call this just before disposing a dialog that uses popup menus.

# Routines for XOPs that Access Files

These routines assist an XOP in opening and saving files and in reading data from and writing data to files in a platform-independent way. For an overview, see **File I/O** on page 267.

Some of these routines use full paths to identify files. The full paths must be “native”. That is, on Macintosh, they must use colons as path separators and on Windows they must use backslashes. Do not use POSIX paths (with forward slashes), even on Mac OS X, because the Carbon routines called by the XOPSupport routines do not support forward slashes.

```
int
MacToWinPath(path)
char path[MAX_PATH_LEN+1];
```

This routine handles file path format conversion for cross-platform XOPs. See **File Path Conversions** on page 268 for background information.

MacToWinPath converts a Macintosh path into a Windows path by replacing ':' with ':\' at the start of a full path and replacing ':' with '\' elsewhere. Also, leading colons are changed to periods. However, it does not change a UNC volume name. Here are examples of the conversion performed by MacToWinPath.

| | | |
|---|---|---|
| C:A:B:C | => | C:\A\B\C |
| \\server\share:A:B:C | => | \\server\share\A\B\C |
| "::FolderA:FileB" | => | "..\FolderA\FileB" |

In the first example, the volume name is "C". In the second example, using a UNC path, it is "\\server\share".

If a Macintosh path contains a '\' character, the resulting path will not work as a Windows path. Therefore, '\' characters must not be used in Mac paths.

If the path is already a Windows path, MacToWinPath does nothing.

**NOTE**: The path may be longer on output than in was on input ('C:' changed to 'C:\' or ':' changed to '.\'). The buffer is assumed to be MAX_PATH_LEN+1 characters long. MacToWin-Path will not overwrite the buffer. It will generate an error if the output path can not fit in MAX_PATH_LEN characters.

When running on an Asian language system, this routine is double-byte-character-aware and assumes that the system default character encoding governs the path.

The function result is 0 if OK or an error code, such as PATH_TOO_LONG.

```
int
WinToMacPath(path)
char path[MAX_PATH_LEN+1];
```

This routine handles file path format conversion for cross-platform XOPs. See **File Path Conversions** on page 268 for background information.

WinToMacPath converts a Windows path into a Macintosh path by replacing ':\' with ':' at the start of a full path and replacing '\' with ':' elsewhere. Also, leading periods are changed to colons. However, it does not change a UNC volume name. Here are examples of the conversion performed by WinToMacPath.

| | | |
|---|---|---|
| C:\A\B\C | => | C:A:B:C |
| \\server\share\A\B\C | => | \\server\share:A:B:C |
| "..\FolderA\FileB" | => | "::FolderA:FileB" |

In the first example, the volume name is "C". In the second example, using a UNC path, it is "\\server\share".

If a Macintosh path contains a '\' character, the resulting path will not work as a Windows path. Therefore, '\' characters must not be used in Mac paths.

If the path is already a Macintosh path, MacToWinPath does nothing.

**NOTE**: The path may be shorter on output than in was on input (':\' or '.\' changed to ':').

When running on an Asian language system, this routine is double-byte-character-aware and assumes that the system default character encoding governs the path.

The function result is 0 if OK or an error code. Currently there is no case in which it returns non-zero.

```
HFSToPosixPath(hfsPath, posixPath[MAX_PATH_LEN+1], isDirectory)
const char* hfsPath;
char posixPath[MAX_PATH_LEN+1];
int isDirectory;
```

Converts an HFS (colon-separated) path into a POSIX (Unix-style) path. This is available only on Mac OS X and only to convert paths into POSIX paths so that we can pass them to the standard file fopen routine or to OS routines that require POSIX paths.

It is allowed for hfsPath and posixPath to point to the same memory.

From the point of view of the Igor user, all paths should be HFS paths although Windows paths are accepted and converted when necessary. POSIX paths are not valid paths in Igor procedures.

When running on an Asian language system, this routine is double-byte-character-aware and assumes that the system default character encoding governs the path.

Returns 0 if OK or an error code. If an error is returned, *posixPath is undefined.

```
int
GetNativePath(path)
char path[MAX_PATH_LEN+1];
```

This routine handles file path format conversion for cross-platform XOPs. See **File Path Conversions** on page 268 for background information.

GetNativePath calls WinToMacPath when running on Macintosh and MacToWinPath when running on Windows. See MacToWinPath and WinToMacPath for details.

When running on an Asian language system, this routine is double-byte-character-aware and assumes that the system default character encoding governs the path.

The function result is 0 if OK or an error code.

```
int
ConcatenatePaths(pathIn1, nameOrPathIn2, pathOut)
const char* pathIn1;
const char* nameOrPathIn2;
char pathOut[MAX_PATH_LEN+1];
```

Concatenates pathIn1 and nameOrPathIn2 into pathOut. pathOut will be a native path. The input paths may use Macintosh or Windows conventions. See **File Path Conversions** on page 268 for background information.

pathIn1 is a full path to a directory. It can end with zero or one separator.

nameOrPathIn2 is either a file name, a folder name or a partial path to a file or folder. It can end with zero or one separator.

pathOut can point to the same memory as either of the input parameters.

The target file or folder does not need to already exist.

For pathIn1, any of the following are legal.

```
"hd:FolderA:FolderB" "hd:FolderA:FolderB:"
"C:\FolderA\FolderB" "C:\FolderA\FolderB\"
```

For nameOrPathIn2, any of the following are legal.

```
"FileA"  "FolderC"
":FolderC"  "\FolderC"
".FolderC" (legal in a Windows path only)
"::FolderC" "\\FolderC"
"..FolderC" (legal in a Windows path only)
"FolderC:FileA"   "FolderC\FileA"
"\FolderC:FileA"  "\FolderC\FileA"
```

Here are some examples of concatenation.

```
"hd:FolderA:FolderB:"   +  "FolderC"   => "hd:FolderA:FolderB:FolderC"
"hd:FolderA:FolderB:"   +  ":FolderC"  => "hd:FolderA:FolderB:FolderC"
"hd:FolderA:FolderB:"   +  "::FolderC" => "hd:FolderA:FolderC"
"C:\FolderA\FolderB\"   +  "FolderC"   => "C:\FolderA\FolderB\FolderC"
"C:\FolderA\FolderB\"   +  "\FolderC"  => "C:\FolderA\FolderB\FolderC"
"C:\FolderA\FolderB\"   +  "\\FolderC" => "C:\FolderA\FolderC"
```

Multiple colons or backslashes in nameOrPathIn2 mean that we want to back up, starting from the folder specified by pathIn1.

When running on an Asian language system, this routine is double-byte-character-aware and assumes that the system default character encoding governs the paths.

The function result is 0 or error code. In case of error, the contents of pathOut is undefined.

```
int
GetDirectoryAndFileNameFromFullPath(fullFilePath, dirPath, fileName)
const char* fullFilePath;
char dirPath[MAX_PATH_LEN+1];
char fileName[MAX_FILENAME_LEN+1];
```

fullFilePath is a full path to a file. It may be a Macintosh path (using colons) or a Windows path (using backslashes).

On output, dirPath is the full native path to the folder containing the file. This path includes a trailing colon (Macintosh) or backslash (Windows).

On output, fileName contains just the name of the file.

The function result is 0 if OK or an error code.

GetDirectoryAndFileNameFromFullPath does not know or care if the file exists or if the directories referenced in the input path exist. It merely separates the file name part from the full path.

A simple implementation of this routine would merely search for colon or backslash characters. However, this simple approach causes problems on Asian systems that use two-byte characters. The problem is that the second byte of a two-byte character may have the same code as a colon or backslash, causing the simple implementation to mistakenly take it for a path separator.

GetDirectoryAndFileNameFromFullPath takes a more complex approach to avoid this problem. To achieve this, the routine has to know the character encoding governing the fullFilePath parameter. GetDirectoryAndFileNameFromFullPath assumes that the system default character encoding governs the fullFilePath parameter.

```
int
FullPathPointsToFile(fullPath)
const char* fullPath;
```

Returns 1 if the path points to an existing file, 0 if it points to a folder or does not point to anything.

fullPath may be a Macintosh or a Windows path.

When running on an Asian language system, this routine is double-byte-character-aware and assumes that the system default character encoding governs the path.

```
int
FullPathPointsToFolder(fullPath)
const char* fullPath;
```

Returns 1 if the path points to an existing folder, 0 if it points to a file or does not point to anything.

fullPath may be a Macintosh or a Windows path.

When running on an Asian language system, this routine is double-byte-character-aware and assumes that the system default character encoding governs the path.

```
int
GetFullMacPathToDirectory(refNum, dirIDOrZero, pathOut, maxPathLen)
long refNum;
long dirIDOrZero;
char* pathOut;
int maxPathLen;
```

This routine is supported on Macintosh only. It is provided only for the benefit of existing Macintosh XOPs. New XOPs use platform-independent techniques and don't require this routine.

If your existing XOP used the PathNameFromDirID or PathNameFromDirWD routines, which were part of several WaveMetrics file-loaders in previous versions of the XOP Toolkit, use GetFullMacPathToDirectory instead.

GetFullMacPathToDirectory stores in pathOut a full path to the folder specified by refNum and dirIDOrZero.

refNum is either a volume reference number or a working directory reference number.

dirIDOrZero must be a directory ID if refNum is a volume reference number.

dirIDOrZero must be zero if refNum is a working directory reference number.

pathOut must be able to hold maxPathLen characters plus the terminating null character.

The function result is 0 if OK or an error code.

```
int
GetLeafName(filePath, leafName)
const char* filePath;
char leafName[MAX_FILENAME_LEN+1];
```

filePath is either "", a valid file name or a valid path to a file.

leafName must be able to hold MAX_FILENAME_LEN+1 bytes.

Returns via leafName the leaf part of the path if it is a path or the contents of filePath if is not a path.

Returns 0 if OK or an error code.

Added in XOP Toolkit 5.04 but works with Igor Pro 3.13 or later

```
int
XOPCreateFile(fullFilePath, overwrite, macCreator, macFileType)
const char* fullFilePath;
int overwrite;
long macCreator;
long macFileType;
```

Creates a file with the location and name specified by fullFilePath.

fullFilePath must be a native path (using colons on Macintosh, backslashes on Windows).

If overwrite is true and a file by that name already exists, it first deletes the conflicting file. If overwrite is false and a file by that name exists, it returns an error.

macFileType is ignored on Windows. On Macintosh, it is used to set the new file's type. For example, use 'TEXT' for a text file.

macCreator is ignored on Windows. On Macintosh, it is used to set the new file's creator code. For example, use 'IGR0' (last character is zero) for an file.

Returns 0 if OK or an error code.

```
int
XOPDeleteFile(fullFilePath)
const char* fullFilePath;
```

Deletes the specified file.

fullFilePath must be a native path (using colons on Macintosh, backslashes on Windows).

Returns 0 if OK or an error code.

```
int
XOPOpenFile(fullFilePath, readOrWrite, fileRefPtr)
const char* fullFilePath;
int readOrWrite;
XOP_FILE_REF* fileRefPtr;
```

If readOrWrite is zero, opens an existing file for reading and returns a file reference via fileRefPtr.

If readOrWrite is non-zero, opens an existing file for writing or creates a new file if none exists and returns a file reference via fileRefPtr.

fullFilePath must be a native path (using colons on Macintosh, backslashes on Windows).

The function result is 0 if OK or an error code.

```
int
XOPCloseFile(XOP_FILE_REF fileRef)
XOP_FILE_REF fileRef;           // Reference returned by XOPOpenFile
```

Closes the referenced file.

Returns 0 if OK or an error code.

```
int
XOPReadFile(fileRef, count, buffer, numBytesReadPtr)
XOP_FILE_REF fileRef;           // Reference returned by XOPOpenFile
unsigned long count;            // Count of bytes to read
void* buffer;                   // Where bytes are stored
unsigned long* numBytesReadPtr; // Output: number of bytes read
```

Reads count bytes from the referenced file into the buffer.

If numBytesReadPtr is not NULL, it stores the number of bytes read in *numBytesReadPtr.

The function result is 0 if OK or an error code.

If bytes remain to be read in the file and you ask to read more bytes than remain, the remaining bytes are returned and the function result is zero. If no bytes remain to be read in the file and you ask to read bytes, no bytes are returned and the function result is FILE_EOF_ERROR.

XOPReadFile is appropriate when you are reading data of variable size, in which case you do not want to consider it an error if the end of file is reached before reading all of the bytes that you requested. If you are reading a record of fixed size, use use XOPReadFile2 instead of XOPReadFile.

```
int
XOPReadFile2(fileRef, count, buffer, numBytesReadPtr)
XOP_FILE_REF fileRef;           // Reference returned by XOPOpenFile
unsigned long count;            // Count of bytes to read
void* buffer;                   // Where bytes are stored
unsigned long* numBytesReadPtr; // Output: number of bytes read
```

Reads count bytes from the referenced file into the buffer.

If numBytesReadPtr is not NULL, it stores the number of bytes read in *numBytesReadPtr.

The function result is 0 if OK or an error code.

If bytes remain to be read in the file and you ask to read more bytes than remain, the remaining bytes are returned and the function result is FILE_EOF_ERROR.

XOPReadFile2 is appropriate when you are reading a record of fixed size, in which case you want to consider it an error if the end of file is reached before reading all of the bytes in the record. If you are reading a record of variable size then you should use XOPReadFile instead of XOPReadFile2.

```
int
XOPReadLine(fileRef, buffer, bufferLength, numBytesReadPtr)
XOP_FILE_REF fileRef;            // Reference returned by XOPOpenFile
void* buffer;                    // Where bytes are stored
unsigned long bufferLength;      // Size of the buffer in bytes
unsigned long* numBytesReadPtr;  // Output: number of bytes read
```

Reads a line of text from the file into the buffer.

buffer points to a buffer into which the line of data is to be read. bufferLength is the size of the buffer. The buffer can hold bufferLength-1 characters, plus the terminating null character.

A line in the file may end with:

```
<end-of-file>
CR
LF
CRLF
```

XOPReadLine reads the next line of text into the buffer and null-terminates it. The terminating CR, LF, or CRLF is not stored in the buffer.

If numBytesReadPtr is not NULL, it stores the number of bytes read in *numBytesReadPtr.

The function result will be LINE_TOO_LONG_IN_FILE if there is not enough room in the buffer to read the entire line. It will be FILE_EOF_ERROR if we hit the end-of-file before reading any characters. It will be zero if we read any characters (even just a CR or LF) before hitting the end of the file.

This routine was designed for simplicity of use. For applications that require blazing speed (e.g., reading files containing tens of thousands of lines or more), a more complex buffering scheme can improve performance considerably.

```
int
XOPWriteFile(fileRef, count, buffer, numBytesWrittenPtr)
XOP_FILE_REF fileRef;               // Reference returned by XOPOpenFile
unsigned long count;                // Count of bytes to write
const void* buffer;                 // Pointer to data to write
unsigned long* numBytesWrittenPtr;  // Output: number of bytes written
```

Writes count bytes from the buffer to the referenced file.

If numBytesWrittenPtr is not NULL, stores the number of bytes written in *numBytesWrittenPtr.

The function result is 0 if OK or an error code.

```
int
XOPGetFilePosition(fileRef, filePosPtr)
XOP_FILE_REF fileRef;            // Reference returned by XOPOpenFile
unsigned long* filePosPtr;
```

Returns via filePosPtr the current file position of the referenced file.

The function result is 0 if OK or an error code.

```
int
XOPSetFilePosition(fileRef, filePos, mode)
XOP_FILE_REF fileRef;            // Reference returned by XOPOpenFile
long filePos;
int mode;
```

Sets the current file position in the referenced file.

If mode is -1, then filePos is relative to the start of the file. If mode is 0, then filePos is relative to the current file position. If mode is 1, then filePos is relative to the end of the file.

The function result is 0 if OK or an error code.

```
int
XOPAtEndOfFile(fileRef)
XOP_FILE_REF fileRef;            // Reference returned by XOPOpenFile
```

Returns 1 if the current file position is at the end of file, 0 if not..

```
int
XOPNumberOfBytesInFile(fileRef, numBytesPtr)
XOP_FILE_REF fileRef;            // Reference returned by XOPOpenFile
unsigned long* numBytesPtr;
```

Returns via numBytesPtr the total number of bytes in the referenced file.

The function result is 0 if OK or an error code.

```
int
XOPOpenFileDialog(prompt, fileFilterStr, indexPtr, initDir, fullFilePath)
const char* prompt;                   // Message displayed in dialog
const char* fileFilterStr;            // Controls types of files shown
int* indexPtr;                        // Controls initial type of file shown
const char* initDir;                  // Sets initial directory
char fullFilePath[MAX_PATH_LEN+1];    // Output path returned here
```

Displays the open file dialog.

Returns 0 if the user chooses a file or -1 if the user cancels or another non-zero number in the event of an error.

Returns the full path to the file via fullFilePath. In the event of a cancel, fullFilePath is unmodified.

fullFilePath is a native path (using colons on Macintosh, backslashes on Windows).

On Windows prompt sets the dialog caption. On Macintosh it sets a prompt string in the dialog.

*fileFilterStr on Macintosh*

If fileFilterStr is "", then the open file dialog displays all types of files, both on Macintosh and Windows. If fileFilterStr is not "", it identifies the type of files to display.

Prior to Carbon, fileFilterStr was a concatenation of Macintosh file type codes. For example, to display text files and Igor Text files, you would pass "TEXTIGTX".

Now this parameter provides control over the Show popup menu which the Macintosh Navigation Manager displays in the Open File dialog. As a consequence, the fileFilterStr is now constructed differently. For example, the string:

```
"Text Files:TEXT,IGTX:.txt,.itx;All Files:****:;"
```

results in two items in the Show popup menu. The first says "Text Files" and displays any file whose Macintosh file type is TEXT or IGTX as well as any file whose file name extension is ".txt" or ".itx". The second item says "All Files" and displays all files.

The two section sections of this fileFilterString are:

```
"Data Files:TEXT,DATA:.txt,.dat,.csv;"
"All Files:****:;"
```

Each section causes the creation of one item in the Show popup menu.

Each section consists of three components: a menu item string to be displayed in the Show popup menu, a list of zero or more Macintosh file types (e.g., TEXT,DATA), and a list of extensions (e.g., .txt,.dat,.csv).

In this example, the first menu item would be "Data Files". When the user selects this menu item, the Open File dialog would show any file whose Macintosh file type is TEXT or DATA plus any file whose extension is .txt, .dat, or .csv.

Note that a colon marks the end of the menu item string, another colon marks the end of the list of Macintosh file types, and a semicolon marks the end of the list of extensions.

The **** file type used in the second section is special. It means that the Open File dialog should display all files. In this section, no extensions are specified because there are no characters between the colon and the semicolon.

The syntax of the fileFilterString is unforgiving. You must not use any extraneous spaces or any other extraneous characters. You must include the colons and semicolons as shown above. The trailing semicolon is required. If there is a syntax error, the entire fileFilterString will be treated as if it were empty, which will display all files.

*fileFilterStr on Windows*

On Windows, fileFilterStr is constructed as for the lpstrFilter field of the OPENFILENAME structure for the Windows GetOpenFileName routine. For example, to allow the user to select text files and Igor Text files, use:

```
"Text Files (*.txt)\0*.txt\0Igor Text Files (*.itx)\0*.itx\0
                                            All Files (*.*)\0*.*\0\0"
```

This would all be on one line in an actual program. Note that the string ends with two null characters (\0\0).

fileIndexPtr is ignored if it is NULL. If it is not NULL, then *fileIndexPtr is the one-based index of the file type filter to be initially selected. In the example given above, setting *fileIndexPtr to 2 would select the Igor Text file filter on entry to the dialog. On exit from the dialog, *fileIndexPtr is set to the index of the file filter string that the user last selected.

initialDir can be "" or it can point to a full path to a directory. It determines the directory that will be initially displayed in the open file dialog. If it is "", the directory will be the last directory that was seen in the open or save file dialogs. If initialDir points to a valid path to a directory, then this directory will be initially displayed in the dialog. initialDir is a native path (using colons on Macintosh, backslashes on Windows).

XOPOpenFileDialog returns via fullFilePath the full path to the file that the user chose or "" if the user cancelled. The path is native path (using colons on Macintosh, backslashes on Windows). fullFilePath must point to a buffer of at least MAX_PATH_LEN+1 bytes.

On Windows, the initial value of fullFilePath sets the initial contents of the File Name edit control in the Open File dialog. The following values are valid:

""

A file name

A full Macintosh or Windows path to a file

On Macintosh, the initial value of fullFilePath is not currently used. It should be set the same as for Windows because it may be used in the future.

In the event of an error other than a cancel, XOPOpenFileDialog displays an error dialog. This should never or rarely happen.

On Windows the dialog will appear in the upper left corner of the screen. This is because Windows provides no straight-forward way to set the position of the dialog.

```
int
XOPSaveFileDialog(prompt, fileFilterStr, indexPtr, initDir, defExt,
                                                          fullFilePath)
const char* prompt;                 // Message displayed in dialog
const char* fileFilterStr;          // Controls types of files shown
int* indexPtr;                      // Controls initial type of file shown
const char* initDir;                // Sets initial directory
const char* defExt;                 // Default file extension
char fullFilePath[MAX_PATH_LEN+1];  // Output path returned here
```

Displays the save file dialog.

Returns 0 if the user provides a file name or -1 if the user cancels or another non-zero number in the event of an error.

Returns the full path to the file via fullFilePath. fullFilePath is both an input and an output as explained below. In the event of a cancel, fullFilePath is unmodified.

fullFilePath is a native path (using colons on Macintosh, backslashes on Windows).

On Windows, prompt sets the dialog caption. On Macintosh, it sets a prompt string in the dialog.

*fileFilterStr on Macintosh*

Prior to Carbon, the fileFilterStr was ignored on Macintosh. You were instructed to pass "" in case it was used in a future version of the XOP Toolkit. Now this parameter is now used to control the contents of the Format popup menu in the Save File dialog.

If there is only one format in which you can save the file, pass "" for fileFilterStr. This will cause the Format menu to be hidden. If you can save the file in more than one format, pass a string like this:

```
"Plain Text:TEXT:.txt;Igor Text:IGTX:.itx;"
```

This would give you a Format menu like this:

```
Plain Text
Igor Text
```

The format of fileFilterStr is the same as the format of the fileFilterStr parameter to XOPNavOpenFileDialog, except that you should specify only one file type and extension for each section.

At present, only the menu item strings ("Plain Text" and "Igor Text" in the example above) are used. The Macintosh file types (TEXT and IGTX) and the extensions (".txt" and ".itx") are currently not used. But you should pass some valid values anyway because a future XOP Toolkit might use them. If there is no meaningful extension, leave the extension section blank.

The Format popup menu in the Save File dialog allows the user to tell you in what format the file should be saved. Unlike the Show popup menu in the Open File dialog, the Format menu has no filtering function. You find out which item the user chose via the fileIndexPtr parameter.

The syntax of the fileFilterStr is unforgiving. You must not use any extraneous spaces or any other extraneous characters. You must include the colons and semicolons as shown above. The trailing semicolon is required. If there is a syntax error, the entire fileFilterStr will be treated as if it were empty, which will display all files.

*fileFilterStr on Windows*

On Windows, fileFilterStr identifies the types of files to display and the types of files that can be created. It is constructed as for the lpstrFilter field of the OPENFILENAME structure for the Windows GetSaveFileName routine. For example, to allow the user to save as a text file or as an Igor Text file, use:

```
"Text Files (*.txt)\0*.txt\0Igor Text Files (*.itx)\0*.itx\0\0"
```

Note that the string ends with two null characters (\0\0). If fileFilterStr is "", this behaves the same as "Text Files (*.txt)\0*.txt\0\0".

fileIndexPtr it is ignored if it is NULL. If it is not NULL, then *fileIndexPtr is the one-based index of the file type filter to be initially selected. In the example given above, setting *fileIndexPtr to 2 would select the Igor Text file type on entry to the dialog. On exit from the dialog, *fileIndexPtr is set to the index of the file type string that the user last selected.

initialDir can be "" or it can point to a full path to a directory. It determines the directory that will be initially displayed in the save file dialog. If it is "", the directory will be the last directory that was seen in the open or save file dialogs. If initialDir points to a valid path to a directory, then this directory will be initially displayed in the dialog. initialDir is a native path (using colons on Macintosh, backslashes on Windows).

defaultExtensionStr is ignored on Macintosh. You must pass "" because this may be used in a future version of the XOP Toolkit.

On Windows, defaultExtensionStr points to the extension to be added to the file name if the user does not enter an extension. For example, pass "txt" to have ".txt" appended if the user does not enter an extension. If you don't want any extension to be added in this case, pass NULL.

XOPSaveFileDialog returns via fullFilePath the full path to the file that the user chose or "" if the user cancelled. The path is a native path (using colons on Macintosh, backslashes on Windows). fullFilePath must point to a buffer of at least MAX_PATH_LEN+1 bytes.

On both Windows and Macintosh, the initial value of fullFilePath sets the initial contents of the File Name edit control in the save file dialog. The following values are valid:

""

A file name

A full Macintosh or Windows path to a file

In the event of an error other than a cancel, XOPSaveFileDialog displays an error dialog. This should never or rarely happen.

On Windows the dialog will appear in the upper left corner of the screen. This is because Windows provides no straight-forward way to set the position of the dialog.

# Routines for File-Loader XOPs

These routines are specialized for file-loader XOPs — XOPs that load data from files into Igor waves.

```
int
GetFullPathFromSymbolicPathAndFilePath(pathName,filePath,fullFilePath)
const char* pathName;                  // Igor symbolic path name
char filePath[MAX_PATH_LEN+1];         // Path to file
char fullFilePath[MAX_PATH_LEN+1];     // Output full path
```

pathName is the name of an Igor symbolic path or "" if no symbolic path is to be used.

filePath is either a full path, a partial path, or a simple file name. It may be a Macintosh path (using colons) or a Windows path (using backslashes).

fullFilePath is an output and will contain the full native path (using colons on Macintosh, backslashes on Windows) to the file referenced by the symbolic path and the file path.

This routine is used by file loader XOPs to get a full native path to a file based on the typical inputs to a file loader, namely an optional Igor symbolic path and an optional full or partial file path or file name.

The two most common cases are:

```
LoadWave <full path to file>
LoadWave/P=<symbolic path name> <file name>
```

where <file name> conotes a simple file name.

Less common cases that this routine also handles are:

```
LoadWave/P=<symbolic path name> <full path to file>
LoadWave/P=<symbolic path name> <partial path to file>
```

In the following cases, the full path to the file can not be determined, so GetFullPathFrom-SymbolicPathAndFilePath returns an error. This would cause a file loader to display an open file dialog:

```
LoadWave <file name>
LoadWave <partial path to file>
```

filePath and fullFilePath may point to the same storage, in which case the output string will overwrite the input string.

This routine does not check that the output path is valid or points to an existing file. This makes the routine useable for applications in which you are creating the file as well as applications in which you are reading the file. If you want to verify that the output path points to an existing file, use the FullPathPointsToFile routine.

The function result is 0 if it was able to create the full path or an error code if not.

```
int
FileLoaderMakeWave(column, waveName, numPoints, fileLoaderFlags, whp)
long column;          // Number of column for this wave
char* waveName;       // Name for this wave
long numPoints;       // Number of points in wave
int fileLoaderFlags; // Standard file-loader flags
waveHndl* whp;        // Place to store handle for new wave
```

This routine is intended for use in simple file-loader XOPs such as SimpleLoadWave. It makes a wave with numPoints points and with the numeric type as specified by fileLoaderFlags.

The function result is 0 or an error code.

Typically you would get fileLoaderFlags using the FileLoaderGetOperationFlags routine.

It returns a handle to the wave via whp or NULL in the event of an error.

fileLoaderFlags is interpreted using the standard file-loader flag bit definitions in XOP.h.

If (fileLoaderFlags & FILE_LOADER_OVERWRITE) is non-zero, FileLoaderMakeWaves overwrites any pre-existing wave in the current data folder with the specified name.

If (fileLoaderFlags & FILE_LOADER_DOUBLE_PRECISION) is non-zero, FileLoaderMakeWaves creates a double-precision floating point wave. Otherwise, it creates a single-precision floating point wave.

If (fileLoaderFlags & FILE_LOADER_COMPLEX) is non-zero, FileLoaderMakeWaves creates a complex wave. Otherwise, it creates a real point wave.

column should be the number of the column being loaded or the number of the wave in a set of waves or zero if this does not apply to your XOP.

**NOTE**: In the event of a name conflict, FileLoaderMakeWave will change the contents of waveName. waveName must be able to hold MAX_OBJ_NAME characters plus a null character.

```
int
SetFileLoaderOutputVariables(fileName, numWavesLoaded, waveNames)
char* fileName;        // Name of the file just loaded
int numWavesLoaded;    // Number of waves loaded
char* waveNames;       // Semicolon-separated list of wave names
```

If your external operation uses Operation Handler, use SetFileLoaderOperationOutputVariables instead of this routine.

SetFileLoaderOutputVariables should be called at the end of a file load to set the standard file loader output globals:

| | |
|---|---|
| S_fileName | The name of the file loaded. |
| S_path | The full path to the folder containing the file. See description below. |
| V_flag | The number of waves loaded. |
| S_waveNames | Semicolon-separate list of wave names (e.g. "wave0;wave1;"). |

fileName can be either just the file name (e.g., "Data File") or a full path including a file name (e.g., "hd:Data Folder:Data File"). If it is a full path, it can be a Macintosh path (using colons) or a Windows path (using backslashes).

If fileName is a full path, SetFileLoaderOutputVariables stores the path to the folder containing the file in S_path and stores the simple file name in S_fileName. In this case, the stored path uses Macintosh path syntax and includes a trailing colon.

If fileName is a simple file name, SetFileLoaderOutputVariables does not set or create S_path and stores the simple file name in S_fileName.

New or updated XOPs should pass the full path to SetFileLoaderOutputVariables so that S_path will be set to a meaningful value.

Returns 0 or an error code.

```
int
SetFileLoaderOperationOutputVariables(runningInUserFunction, fileName,
                                        numWavesLoaded, waveNames)
int runningInUserFunction; // Truth that operation was called from
function
const char* fileName;      // Name of file loaded or full path to file
int numWavesLoaded;        // Number of waves created
const char* waveNames;     // Semicolon-separated list of wave names
```

This function does the same thing as SetFileLoaderOutputVariables except that, when runningInUserFunction is true, it sets local variables in the user function. You obtain the value to pass for the runningInUserFunction parameter from the calledFromFunction field of your operation's runtime parameter structure.

This function is intended to be used only when you are implementing an external operation using Operation Handler. When you register your operation via RegisterOperation, you must specify that your operation sets the numeric variable V_flag and the string variables S_fileName, S_path, and S_waveNames. See SimpleLoadWaveOperation.c for an example.

See discussion of SetFileLoaderOutputVariables for further details.

Returns 0 or error code.

Added in Igor Pro 5.0 but works with any version. If you call this when running with an earlier version, it acts just like SetFileLoaderOutputVariables regardless of the value of runningInUserFunction.

# Routines for XOPs with Windows

These routines provide support for XOPs that add a window to Igor. See Chapter 9, **Adding Windows**, for further information.

```
WindowPtr
GetXOPWindow(windowID, wStorage, behind)
int windowID;          // ID for WIND resource in XOP's resource fork
Ptr wStorage;          // Place to store window record or NULL
WindowPtr behind;      // Window behind which new window should go or -1
```

This routine is supported on Macintosh only. See Chapter 9 for a discussion of creating a window in a Windows XOP.

Opens a new window for an XOP.

The parameters and result are the same as for the Macintosh Toolbox GetNewCWindow call.

GetXOPWindow sets the windowKind field of the new window so that Igor can recognize it as belonging to your XOP.

```
XOP_WINDOW_REF
GetActiveWindowRef(void)
```

Returns an XOP_WINDOW_REF for the active window.

An XOP_WINDOW_REF is a WindowPtr on Macintosh and an HWND on Windows. The returned value could be a reference to a window that is not owned by the calling XOP.

```
int
IsXOPWindowActive(windowRef)
XOP_WINDOW_REF windowRef;
```

Returns true if the specified window is the active window, false if not.

An XOP_WINDOW_REF is a WindowPtr on Macintosh and an HWND on Windows.

```
void
ShowAndActivateXOPWindow(windowRef)
XOP_WINDOW_REF windowRef;
```

Shows the specified window if it is hidden and then makes it the active window.

An XOP_WINDOW_REF is a WindowPtr on Macintosh and an HWND on Windows.

```
void
HideAndDeactivateXOPWindow(windowRef)
XOP_WINDOW_REF windowRef;
```

Deactivates the window if it is the active window and then hides it.

An XOP_WINDOW_REF is a WindowPtr on Macintosh and an HWND on Windows.

```
void
SetXOPWindowTitle(windowRef, title)
XOP_WINDOW_REF windowRef;
const char* title;
```

Sets the title (also known as "caption") for the window to the string specified by title.

```
void
GetXOPWindowPositionAndState(windowRef, r, winStatePtr)
XOP_WINDOW_REF windowRef;
Rect* r;                    // Receives window's coordinates.
int* winStatePtr;           // Receives window state bits.
```

Returns the window's position on the screen in pixels and its state. Use this with SetXOPWindowPositionAndState to save and restore a window's position and state.

Use this routine when you need to store a window position in a platform-dependent way, for example, in a preference file. Use GetXOPWindowIgorPositionAndState to store a window position in a platform-independent way, for example, in a /W=(left,top,right,bottom) flag.

On Macintosh, the returned coordinates specify the location of the window's content region in global coordinates. Bit 0 of *winStatePtr is set if the window is visible and cleared if it is hidden. All other bits are set to 0.

On Windows, the returned coordinates specify the the location of the entire window in its normal state relative the the top/left corner of the Igor MDI client window. Bit 0 of *winStatePtr is set if the window is visible and cleared if it is hidden. Bit 1 of *winStatePtr is set if the window is minimize and cleared if it is not minimized. All other bits are set to 0.

On either platform, the returned rectangle is a Macintosh rectangle.

```
void
SetXOPWindowPositionAndState(windowRef, r, winStatePtr)
XOP_WINDOW_REF windowRef;
Rect* r;                    // Contains window's coordinates.
int* winStatePtr;           // Contains window state bits.
```

Moves the XOP window to the position indicated by r and sets its state. Use this with GetXOPWindowPositionAndState to save and restore a window's position and state.

Use this routine when you need to restore a window position in a platform-dependent way, for example, in a preference file. Use SetXOPWindowIgorPositionAndState to restore a window position in a platform-independent way, for example, in a /W=(left,top,right,bottom) flag.

See GetXOPWindowPositionAndState for a discussion of the units of the rectangle and the meaning of the winState parameter.

This routine makes an effort to prevent the window from becoming inaccessible because it is off-screen.

```
void
TransformWindowCoordinates(mode, coords)
int mode;
double coords[4];
```

Transforms window coordinates from screen pixels into Igor coordinates or from Igor coordinates into screen pixels.

This routine is intended for use in command line operations that set a window position, for example, for an operation that supports a /W=(left,top,right,bottom) flag. We want a given command containing a /W flag to produce approximately the same result on Macintosh and on Windows. This is complicated because of differences in the way each platform represents the position of windows.

Igor coordinates are a special kind of coordinates that were designed to solve this problem. They are described in the section **Igor Window Coordinates** on page 255.

mode is

0: Transform from screen pixels into Igor coordinates.
1: Transform from Igor coordinates into screen pixels.

For TransformWindowCoordinates, screen pixels are in global coordinates on Macintosh (relative to the top/left corner of the main screen) and are in MDI-client coordinates (relative to the top/left corner of the MDI client window, not the MDI frame) on Windows.

coords is an array of window coordinates. It is both an input and an output. The coordinates specify the location of the window's content region only. That is, it excludes the title bar and the frame.

coords[0] is the location of the left edge of the window content region.

coords[1] is the location of the top edge of the window content region.

coords[2] is the location of the right edge of the window content region.

coords[3] is the location of the bottom edge of the window content region.

On Macintosh, screen coordinates and Igor coordinates are identical. Thus, this routine is a NOP on Macintosh.

```
void
GetXOPWindowIgorPositionAndState(windowRef, coords, winStatePtr)
XOP_WINDOW_REF windowRef;
double coords[4];
int* winStatePtr;
```

Returns the XOP window's position on the screen in Igor coordinates and its state. Use this with SetXOPWindowIgorPositionAndState to save and restore a window's position and state.

Use this routine when you need to store a window position in a platform-independent way, for example, in a /W=(left,top,right,bottom) flag. Use GetXOPWindowPositionAndState to store a window position in a platform-dependent way, for example, in a preference file.

See **Igor Window Coordinates** on page 255 for a discussion of Igor coordinates.

On both Macintosh and Windows, the returned coordinates specify the location of the window's content region, not the outside edges of the window. On Windows, the returned coordinates specify the the location of the window in its normal state even if the window is minmized or maximized.

On Macintosh, bit 0 of *winStatePtr is set if the window is visible and cleared if it is hidden. All other bits are set to 0.

On Windows, bit 0 of *winStatePtr is set if the window is visible and cleared if it is hidden. Bit 1 of *winStatePtr is set if the window is minimize and cleared if it is not minimized. All other bits are set to 0.

```
void
SetXOPWindowIgorPositionAndState(windowRef, coords, winState)
XOP_WINDOW_REF windowRef;
double coords[4];
int winState;
```

Moves the XOP window to the position indicated by coords and sets its state. Use this with GetXOPWindowIgorPositionAndState to save and restore a window's position and state.

Use this routine when you need to restore a window position in a platform-independent way, for example, in a /W=(left,top,right,bottom) flag. Use SetXOPWindowPositionAndState to restore a window position in a platform-dependent way, for example, in a preference file.

See **Igor Window Coordinates** on page 255 for a discussion of Igor coordinates.

On both Macintosh and Windows, the coordinates must specify the location of the window's content region, not the outside edges of the window. On Windows, the coordinates specify the the location of the window in its normal state even if the window is minmized or maximized.

On Macintosh, bit 0 of winState is set if the window is visible and cleared if it is hidden. All other bits are set to 0.

On Windows, bit 0 of winState is set if the window is visible and cleared if it is hidden. Bit 1 of winState is set if the window is minimize and cleared if it is not minimized. All other bits are set to 0.

This routine makes an effort to prevent the window from becoming inaccessible because it is off-screen.

```
void
ArrowCursor(void)
```

Sets the cursor to the arrow.

Your XOP can call this if it owns the top window.

```
void
IBeamCursor(void)
```

Sets the cursor to the I beam.

Your XOP can call this if it owns the top window.

```
void
HandCursor(void)
```

Sets the cursor to the hand.

Your XOP can call this if it owns the top window.

```
void
WatchCursor(void)
```

Sets the cursor to the watch.

Your XOP can call this if it is doing a time-consuming operation.

```
void
SpinCursor(void)
```

Sets the cursor to the spinning beachball.

Your XOP can call this if it is doing a time-consuming operation.

To spin the beachball cursor and also allow background processing, call SpinProcess.

# Routines for XOPs with Text Windows

Igor provides a group of routines, called TU ("Text Utility") routines, that allow an XOP to implement a fully functional text window, like Igor's built-in procedure window. The sample XOP TUDemo illustrates using TU routines. There is a discussion of TU windows on page 254.

## Creating and Disposing Text Windows

```
int
TUNew2(winTitle, winRectPtr, TUPtr, windowRefPtr)
const char* winTitle;
const Rect* winRectPtr;
TUStuffHandle* TUPtr;
XOP_WINDOW_REF* windowRefPtr;
```

winTitle points to the title (also known as "caption") for the new window.

winRectPtr points to a Macintosh Rect defining the location and size of the window in units of pixels. The Rect defines the "content region" of the window - that is, the are exclusive of the title bar or caption and frame.

On Macintosh, the rectangle is in global coordinates. Use a top coordinate of 40 to position the window right below the menu bar.

On Windows, the rectangle is in client window coordinates of the Igor MDI frame window. Use a top coordinate of 22 to position the window right below the menu bar.

TUNew2 returns via TUPtr a handle to the TU document. You pass this handle back to Igor when calling TU XOPSupport routines.

TUNew2 also returns via windowRefPtr a pointer to a WindowPtr (Mac) or HWND (Windows) for the newly created window.

In the event of an error, it returns non-zero as the function result and NULL via TUPtr and windowRefPtr. In the event of success, it returns zero as the function result.

TUNew2 uses a default font and font size. The resulting text document is like an Igor plain text notebook.

The window is initially hidden. Call ShowAndActivateXOPWindow to show it.

```
TUStuffHandle
TUNew(winPtr, borderRectPtr, font, size, crOnly)
WindowPtr winPtr;     // Pointer to window record for text window
Rect* borderRectPtr; // Pointer to rect defining text area in window
int font, size;      // Font number and type size to use for text
int crOnly;          // -1 for no wrap around, 0 for wrap around
```

**NOTE**: TUNew is available to Macintosh XOPs only. New cross-platform XOPs should use TUNew2 for both Macintosh and Windows.

TUNew returns a handle to a record containing all information about the text area.

Pass NULL for winPtr if you want TUNew to create a new window.

**NOTE**: The borderRectPtr parameter is not implemented. The entire window will be used for text. Pass NULL for this parameter.

**NOTE**: The text utility/XOP interface can not presently handle wrapping text so crOnly should always be -1.

```
void
TUDispose(TU)
TUStuffHandle TU;          // Handle to information about the text area
```

Disposes all data structures associated with TU.

Call this when your XOP is cleaning up before being closed.

If you created the window by calling TUNew2, then TUDispose always disposes the window.

If you created the window by calling the older TUNew routine, then TUDispose disposes it only if you did *not* pass NULL as the winPtr parameter to TUNew. If you did pass NULL, then you must dispose of the window yourself after calling TUDispose.

## Responding to Text Window Messages

```
void
TUDisplaySelection(TU)
TUStuffHandle TU;          // Handle to information about the text area
```

Scrolls the specified text area until the selected text is in view.

Call this in response to the DISPLAYSELECTION message from Igor.

```
void
TUGrow(TU, size)
TUStuffHandle TU;    // Handle to information about the text area
long size;           // Vertical in high word, horizontal in low word
```

Grows the window containing the text area according to size.

Macintosh XOPs must call this routine in response to the GROW message from Igor.

Windows XOPs don't receive the GROW message. Instead, it is handled internally in Igor. Therefore, Windows XOPs do not need to call this routine.

If size is zero, it zooms the window in or out.

If size is –1, this tells TUGrow to adjust the TU window to a change in window size that has already been done. This causes TUGrow to adjust to the change in size, for example, to move the scroll bars to their new position, after the XOP has resized the window.

```
void
TUUpdate(TU)
TUStuffHandle TU;          // Handle to information about the text area
```

Updates the TU window if its update region is not empty.

Macintosh XOPs must call this routine in response to the UPDATE message from Igor.

Windows XOPs don't receive the UPDATE message. Instead, it is handled internally in Igor. Therefore, Windows XOPs do not need to call this routine.

```
void
TUActivate(TU, flag)
TUStuffHandle TU;          // Handle to information about the text area
int flag;                  // Flag = 0 for deactivate, 1 for activate
```

Activates or deactivates the text area.

Macintosh XOPs must call this routine in response to the ACTIVATE message from Igor.

Windows XOPs don't receive the ACTIVATE message. Instead, it is handled internally in Igor. Therefore, Windows XOPs do not need to call this routine.

```
void
TUIdle(TU)
TUStuffHandle TU;          // Handle to information about the text area
```

Blinks the caret in the text area.

Call this in response to the IDLE message from Igor.

```
void
TUMoveToPreferredPosition(TU)
TUStuffHandle TU;          // Handle to information about the text area
```

Moves the window to the preferred position, as determined by the user's notebook preferences. Normally, you will call this in response to the MOVE_TO_PREFERRED_POSITION message from Igor.

During the TUMoveToPreferredPosition call, your XOP may receive GROW, WINDOW_MOVED, and possibly other message from Igor.

```
void
TUMoveToFullSizePosition(TU)
TUStuffHandle TU;          // Handle to information about the text area
```

Moves the window to show all of its content or to fill the screen (Macintosh) or MDI frame window (Windows). Normally, you will call this in response to the MOVE_TO_FULL_POSITION message from Igor.

During the TUMoveToFullSizePosition call, your XOP may receive GROW, WINDOW_MOVED, and possibly other message from Igor.

```
void
TURetrieveWindow(TU)
TUStuffHandle TU;          // Handle to information about the text area
```

Moves the window, if necessary, to fit entirely within the screen (Macintosh) or MDI frame window (Windows). Normally, you will call this in response to the RETRIEVE message from Igor.

During the TURetrieveWindow call, your XOP may receive GROW, WINDOW_MOVED, and possibly other message from Igor.

```
void
TUNull(TU)
TUStuffHandle TU;          // Handle to information about the text area
```

Sets the cursor according to the position of mouse.

Macintosh XOPs must call this routine in response to the NULLEVENT message from Igor.

Windows XOPs don't receive the NULLEVENT message. Instead, it is handled internally in Igor. Therefore, Windows XOPs do not need to call this routine.

```
void
TUCopy(TU)
TUStuffHandle TU;          // Handle to information about the text area
```

Copies the selected text to clipboard.

Call this in response to the COPY message from Igor.

```
void
TUCut(TU)
TUStuffHandle TU;          // Handle to information about the text area
```

Cuts the selected text to clipboard.

Call this in response to the CUT message from Igor.

```
void
TUPaste(TU)
TUStuffHandle TU;           // Handle to information about the text area
```

Pastes text from the clipboard into the text area.

Call this in response to the PASTE message from Igor.

```
void
TUClear(TU)
TUStuffHandle TU;           // Handle to information about the text area
```

Clears the selected text.

Call this in response to the CLEAR message from Igor.

```
void
TUUndo(TU)
TUStuffHandle TU;           // Handle to information about the text area
```

Undoes the previous action.

Call this in response to the UNDO message from Igor.

```
void
TUPrint(TU)
TUStuffHandle TU;           // Handle to information about the text area
```

Allows the user to print the selected text or all of text if there is no selection.

Call this in response to the PRINT message from Igor.

```
void
TUPageSetupDialog(TU)
TUStuffHandle TU;           // Handle to information about the text area
```

Displays a page setup dialog.

Call this in response to the PAGESETUP message from Igor.

```
void
TUClick(TU)
TUStuffHandle TU;           // Handle to information about the text area
```

Handles clicks in the text area.

Macintosh XOPs must call this routine in response to the CLICK message from Igor.

Windows XOPs don't receive the CLICK message. Instead, it is handled internally in Igor. Therefore, Windows XOPs do not need to call this routine.

```
void
TUKey(TU, eventPtr)
TUStuffHandle TU;          // Handle to information about the text area
EventRecord* eventPtr;     // Pointer to keydown event record
```

Handles the keystroke.

Macintosh XOPs must call this routine in response to the KEY message from Igor.

Windows XOPs don't receive the KEY message. Instead, it is handled internally in Igor. Therefore, Windows XOPs do not need to call this routine.

```
void
TUSelectAll(TU)
TUStuffHandle TU;          // Handle to information about the text area
```

Selects all of the text in the text area.

Call this in response to the SELECT_ALL message from Igor.

```
void
TUFind(TU, code)
TUStuffHandle TU;          // Handle to information about the text area
int code;                  // 1= normal, 2= find same, 3= find selection
```

Allows the user to find text in the text area.

Call this in response to the FIND message from Igor.

```
void
TUReplace(TU)
TUStuffHandle TU;          // Handle to information about the text area
```

Allows the user to replace text in the text area.

Call this in response to the REPLACE message from Igor.

```
void
TUIndentLeft(TU)
TUStuffHandle TU;          // Handle to information about the text area
```

Indents the selected text left by one tab stop.

Call this in response to the INDENTLEFT message from Igor.

```
void
TUIndentRight(TU)
TUStuffHandle TU;          // Handle to information about the text area
```

Indents the selected text right by one tab stop.

Call this in response to the INDENTRIGHT message from Igor.

```
void
TUFixEditMenu(TU)
TUStuffHandle TU;           // Handle to information about the text area
```

Enables or disables items in the Edit menu according to state of the text area.

Call this in response to the MENUENABLE message from Igor.

```
void
TUFixFileMenu(TU)
TUStuffHandle TU;           // Handle to information about the text area
```

Enables or disables the items in File menu according to the state of text area.

Call this in response to the MENUENABLE message from Igor.

```
int
TUSFInsertFile(TU, prompt, fileTypes, numTypes)
TUStuffHandle TU;           // Handle to information about the text area
char* prompt;               // C string for prompt in open dialog
OSType fileTypes[];         // Array of file types that you can open
short numTypes;             // Number of file types in fileTypes[]
```

Allows the user to insert text from a file into your document.

It puts up the open file dialog to allow the user to choose a file.

Call this in response to the INSERTFILE message from Igor.

The function result is 0 if OK or an error code.

```
int
TUSFWriteFile(TU, prompt, fileType, allFlag)
TUStuffHandle TU;           // Handle to information about the text area
char* prompt;               // C string for prompt in save dialog
OSType fileType;            // File type to save file as
short allFlag;              // 0= write selected text, 1= write all text
```

Allows the user to save text from the text area in a file.

It puts up the save file dialog to allow the user to specify the file.

Call this in response to the SAVEFILE message from Igor.

The SAVEFILE message is obsolete. See Chapter 4 for details.

The function result is 0 if OK or an error code.

## Text Window Utility Routines

The rest of the text utility routines are not tied to any specific message from Igor. Rather, they allow you to manage the window and manipulate the text programmatically as necessary for your XOP.

```
void
TUDrawWindow(TU)
TUStuffHandle TU;           // Handle to information about the text area
```

Draws the text and scroll bars in the text area.

```
long
TULines(TU)
TUStuffHandle TU;           // Handle to information about the text area
```

Returns the total number of lines of text in the text area.

```
int
TUGetDocInfo(TU, dip)
TUStuffHandle TU;           // Handle to information about the text area
TUDocInfoPtr dip;           // Receives info about the document
```

This routine provides a way for you to get miscellaneous information about the TU document. At present, it is mostly useful for finding the number of paragraphs in the document. The TUDocInfo structure is defined in IgorXOP.h.

TUGetDocInfo returns information about the text utility document via the structure pointed to by dip. You set the version field of the TUDocInfo structure before calling TUGetDocInfo so that Igor knows which version of the structure your XOP is using.

The function result is 0 if OK, -1 if the version of Igor that is running does not support the doc info version or an error code if the version of Igor that is running does not support this callback.

***Example***

```
TUDocInfo di;
long paragraphs;
int result;

di.version = TUDOCINFO_VERSION;
if (result = TUGetDocInfo(theTU, &di))
   return result;
paragraphs = di.paragraphs;
```

```
int
TUGetSelLocs(TU, startLocPtr, endLocPtr)
TUStuffHandle TU;          // Handle to information about the text area
TULocPtr startLocPtr;      // Receives location of start of selection
TULocPtr endLocPtr;        // Receives location of end of selection
```

Sets *startLocPtr and *endLocPtr to describe the selected text in the document.

The TULoc structure is defined in IgorXOPs.h. A text location consists of a paragraph number, starting from zero, and a character position starting from zero.

```
int
TUSetSelLocs(TU, startLocPtr, endLocPtr, flags)
TUStuffHandle TU;          // Handle to information about the text area
TULocPtr startLocPtr;      // Contains location of start of selection
TULocPtr endLocPtr;        // Contains location of end of selection
int flags;                 // Miscellaneous flags
```

If startLocPtr is not NULL, sets the selection in the text area based on startLocPtr and endLocPtr which must be valid.

If flags is 1, it displays the selection if it is out of view. Other bits in flags may be used for other purposes in the future. For now they should all be 0.

The function result is 0 if OK, an error code if the TULocs are not valid or if the version of Igor that is running does not support this callback.

For the TULoc structure to be valid, its paragraph field must be between 0 and p-1 where p is the number of paragraphs in the document as reported by TUGetDocInfo. The character position field must be between 0 and c where c is the number of characters in the paragraph as reported by TUFetchParagraphText. In a paragraph with c characters, the location before the first character corresponds to a position of zero, the location before the last character corresponds to a position of c-1 and the location after the last character corresponds to a position of c.

```
void
TUInsert(TU, dataPtr, dataLen)
TUStuffHandle TU;          // Handle to information about the text area
char* dataPtr;             // Pointer to text to insert
long dataLen;              // Number of characters to insert
```

Inserts the specified text in the text area.

This allows your XOP to insert text at any time.

```
void
TUDelete(TU)
TUStuffHandle TU;           // Handle to information about the text area
```

Deletes the selected text in the text area.

This allows your XOP to delete text at any time.

```
int
TUInsertFile(TU, fileName, wdRefNum)
TUStuffHandle TU;           // Handle to information about the text area
char* fileName;             // C string containing name of file to insert
short wdRefNum;             // Working directory refNum for file
```

Inserts text from the file at the insertion point in the text area.

This is handy for reloading text that you saved as part of an experiment.

The function result is 0 if OK or an error code.

```
int
TUWriteFile(TU, fileName, wdRefNum, allFlag)
TUStuffHandle TU;           // Handle to information about the text area
char* fileName;             // C string name of file to write
short wdRefNum;             // Working directory refNum for file
short allFlag;              // 0= write selected text, 1= write all text
```

Writes text from the text area to the specified file.

This is handy for saving text as part of an experiment.

The function result is 0 if OK or an error code.

```
int
TUFetchParagraphText(TU, paragraph, textPtrPtr, lengthPtr)
TUStuffHandle TU;     // Handle to information about the text area
long paragraph;       // Number of paragraph to fetch starting from 0
char* textPtrPtr;     // Receives pointer to text
long* lengthPtr;      // Receives number of characters in paragraph
```

This routine fetches all of the text in the specified paragraph.

If textPtrPtr is not NULL, it returns via textPtrPtr a pointer to the text in the specified paragraph.

Sets *lengthPtr to the number of characters in the paragraph whether textPtrPtr is NULL or not.

textPtrPtr is a pointer to your char* variable. Igor allocates a pointer, using NewPtr, and sets *textPtrPtr to point to the allocated memory. You should dispose this when you no longer need it.

The function result is 0 if OK, an error code if the paragraph is out of range or if an error occurs fetching the text or if the version of Igor that is running does not support this callback.

***Example***

```
char* p;
long paragraph, length;
int result;

paragraph = 0;
if (result = TUFetchParagraphText(TU, paragraph, &p, &length))
   return result;
<Deal with the text pointed to by p>
DisposePtr(p);
```

Note that the text pointed to by p is not null terminated and therefore is not a C string.

```
int
TUFetchSelectedText(TU, textHandlePtr, reservedForFuture, flags)
TUStuffHandle TU;           // Handle to information about the text area
Handle* textHandlePtr;      // Receives handle containing text
void* reservedForFuture;    // Pass NULL for this
long flags;                 // Miscellaneous flags
```

Returns via textHandlePtr the selected text in the text utility document.

textHandlePtr is a pointer to your Handle variable. Igor allocates a handle, using NewHandle, and sets *textHandlePtr to refer to the allocated memory. You should dispose this when you no longer need it.

Note that the text in the handle is *not* null terminated. Use GetHandleSize to find the number of bytes in the handle. To use C string functions on this text you need to copy it to a local buffer and null-terminate it or add a null terminator to the handle and lock the handle. If you pass the handle back to Igor, you must remove the null terminator and unlock the handle. See **Understand the Difference Between a String in a Handle and a C String** on page 321.

reservedForFuture should be NULL for now.

flags should be 0 for now.

The function result is 0 if OK, an error code if an error occurs fetching the text or if the version of Igor that is running does not support this callback.

***Example***

```
Handle h;
int result;

paragraph = 0;
if (result = TUFetchSelectedText(TU, &h, NULL, 0))
   return result;
<Deal with the text in handle>
DisposeHandle(h);
```

```
int
TUSetStatusArea(TU, message, eraseFlags, statusAreaWidth)
TUStuffHandle TU;           // Handle to information about the text area
char* message;              // Message to display or NULL
int eraseFlags;             // Determine when message will be erased
statusAreaWidth;            // Desired width of status area
```

Controls the status area in the bottom/left-hand corner of an Igor Pro text window.

If message is not NULL, it sets the status message in window. message is a C string. Only the first 127 characters are displayed.

If message is not NULL then eraseFlags determines when the status message will be erased. See the TU_ERASE_STATUS #defines in IgorXOP.h.

If statusAreaWidth is >= 0, it sets the width of the status area. This is in pixels. Pass -1 if you don't want to change the status area width.

The function result is 0 if OK, an error code if the version of Igor that is running does not support this callback.

# Routines for Dealing with Resources

These callbacks are useful for accessing resources from within an XOP. See **Macintosh Programming Issues** on page 288 for a discussion of issues involving resources on Macintosh.

```
int
XOPRefNum(void)
```

This routine is supported on Macintosh only. There is no Windows equivalent.

Returns the file reference number for the XOP's own resource fork.

```
Handle
GetXOPResource(resType, resID)
long resType;              // Resource type, e.g., 'DLOG'
int resID;                 // Resource ID number
```

This routine is supported on Macintosh only. There is no Windows equivalent.

GetXOPResource returns a handle to the specified resource in the XOP's resource fork or NULL if there is no such resource.

This routine does not search any other resource forks.

```
Handle
GetXOPNamedResource(resType, name)
long resType;              // Resource type, e.g., 'DLOG'
char* name;                // C string containing resource name
```

This routine is supported on Macintosh only. There is no Windows equivalent.

GetXOPNamedResource returns a handle to the specified resource in the XOP's resource fork or NULL if there is no such resource.

This routine does not search any other resource forks.

```
void
GetXOPIndString(text, strID, item)
char* text;          // C string to hold up to 256 characters
int strID;           // ID of STR# resource in XOP's resource fork
int item;            // String number within resource starting from one
```

Returns the specified string from the specified STR# resource in the XOP's resource fork via the text parameter. This routine does not search any other resource forks.

This routine can be used on both Macintosh and Windows.

# Routines for XOPs That Use FIFOs

Igor Pro provides First-In-First-Out (FIFO) objects for use in data acquisition tasks where a continuous stream of data is generated and where you want to monitor the data in real-time with a visual chart recorder-like display. If you have a continuous stream of data but you don't want to use a chart display to monitor it, then you can do your own thing and you do not need to involve FIFOs.

There are two ways to get acquired data from an XOP into a FIFO.

If the data rate is slow, say less than 10 values/second, then you could set up an Igor Pro background task to call your XOP to get data. In this case your XOP does not need to use the FIFO routines. There are problems associated with this technique but it is *much* easier than directly writing into FIFOs. See the Igor Pro manual for a discussion of background tasks.

If the data is coming in quickly, then you will need to directly write data into an Igor Pro FIFO object. You can get a feel for the FIFO technique by examining the Sound Chart Demo example experiment ("Igor Pro Folder:Examples:Movies & Audio:Sound Input").

To use the FIFO technique, you will need to set up an idle routine in your XOP that gets the handle to a given FIFO object, writes data to it and then notifies Igor Pro of that fact. Since idle routines are only called when Igor Pro "gets a chance", you will have to define your own buffer to store data until the next time your idle routine is called. Under some conditions, it can be many seconds between idle calls. Your buffer has to be large enough to handle this amount of data.

If you decide to use a FIFO, you can contact WaveMetrics to get the source code for the SoundInput XOP. This old Mac OS 9 XOP has been superceded by the built-in SoundInRecord operation but it can still serve as an example.

In the SoundInput XOP, a pointer based FIFO pre-buffer is defined as struct PtrFIFO which is defined in the NamedFIFO.h file. This buffer is allocated when an Igor Pro procedure calls the SIStartChart external operation in the XOP. The buffer is filled with sound data via the interrupt completion routine SndCompletion. The idle routine, IdleFIFO, transfers data from the pre-buffer to the Igor Pro FIFO. IdleFIFO calls GetCheckFIFO to get the handle to the FIFO and to perform sanity checks. GetCheckFIFO in turn calls the XOPSupport routine GetNamedFIFO to fetch the FIFO handle. IdleFIFO then transfers data from the pre-buffer to the FIFO and calls the XOPSupport routine MarkFIFOUpdated to notify Igor Pro that new data has been put in the FIFO.

There is additional documentation for FIFOs in the Igor Pro manual. XOP support for FIFOs consists of the following routines plus the NamedFIFO.h file in the XOPSupport folder.

```
struct NamedFIFO **
GetNamedFIFO(name)
char name[MAX_OBJ_NAME+1];        // C string to receive name
```

Returns the named FIFO handle or NULL if none of that name exists.

FIFO handles belong to Igor so you should not dispose them.

```
void
MarkFIFOUpdated(fifo)
struct NamedFIFO **fifo;   // Handle to an existing FIFO
```

Tells Igor Pro that you have modified the data in a FIFO.

Call this after putting data in a named FIFO so that Igor Pro will refresh chart gadgets.

# Numeric Conversion Routines

These XOPSupport routines convert between various numeric formats and are useful for importing or exporting data as well as for other purposes. They are defined in the file XOPNumericConversion.c.

Some of the routines take a pointer to some input data and a pointer to a place to put the converted output data. Numbers can be converted in place. That is, the pointer to the input data and the pointer to the output data can point to the same array in memory. Make sure that the array is big enough to hold the data in the output format.

These routines are used in the GBLoadWave, VDT2 and NIGPIB2 sample XOPs.

XOP Toolkit routines use two different ways to specify a number type:

- XOP Toolkit number format code plus a count of bytes per number.
- Igor number type code.

This table shows how these two methods relate:

| Number Type | XOP Toolkit Code/Count | Igor Code |
|---|---|---|
| Double-precision floating point | IEEE_FLOAT, 8 | NT_FP64 |
| Single-precision floating point | IEEE_FLOAT, 4 | NT_FP32 |
| Long integer (32 bits) | SIGNED_INT, 4 | NT_I32 |
| Short integer (16 bits) | SIGNED_INT, 2 | NT_I16 |
| Byte integer (8 bits) | SIGNED_INT, 1 | NT_I8 |
| Unsigned long integer (32 bits) | UNSIGNED_INT, 4 | NT_I32 \| NT_UNSIGNED |
| Unsigned short integer (16 bits) | UNSIGNED_INT, 2 | NT_I16 \| NT_UNSIGNED |
| Unsigned byte integer (8 bits) | UNSIGNED_INT, 1 | NT_I8 \| NT_UNSIGNED |

Originally, Igor supported only single-precision (NT_FP32) and double-precision (NT_FP64) floating point. Thus, to describe other types, the XOP Toolkit codes were invented.

```
int
ConvertData(src,dest,numValues,srcBytes,srcFormat,destBytes,destFormat)
void* src;            // Input data in format specified by srcFormat
void* dest;           // Output data in format specified by destFormat
long numValues;       // # of values to convert
int srcBytes;         // # of bytes in a point of input (1, 2, 4, or 8)
int srcFormat;        // XOP Toolkit code for input data format
int destBytes;        // # of bytes in a point of output (1, 2, 4, or 8)
int destFormat;       // XOP Toolkit code for output data format
```

This routine calls the appropriate conversion routine based on the srcBytes, srcFormat, destBytes and destFormat parameters.

ConvertData can handle any combination of the legal Igor numeric data types. It takes the description of source and destination data types in the form of an XOP Toolkit numeric format code, defined in XOPSupport.h, and a number of bytes. Both the source and destination can be any of the following:

```
format = IEEE_FLOAT, numBytes = 8       // Double-precision IEEE float
format = IEEE_FLOAT, numBytes = 4       // Single-precision IEEE float
format = SIGNED_INT, numBytes = 4       // 32 bit signed integer
format = SIGNED_INT, numBytes = 2       // 16 bit signed integer
format = SIGNED_INT, numBytes = 1       // 8 bit signed integer
format = UNSIGNED_INT, numBytes = 4     // 32 bit unsigned integer
format = UNSIGNED_INT, numBytes = 2     // 16 bit unsigned integer
format = UNSIGNED_INT, numBytes = 1     // 8 bit unsigned integer
```

Returns zero if everything is OK, 1 if the conversion is not supported (e.g., converting to 2 byte floating point), -1 if no conversion is needed (source format == dest format).

ConvertData relies on a large number of low-level conversion routines with names of the form <Source Type>To<Dest Type> where both <Source Type> and <Dest Type> can be any of the following:

Double, Float, Long, Short, Byte, UnsignedLong, UnsignedShort, UnsignedByte

For most uses, you should use the high level ConvertData routine.

```
int
ConvertData2(src, dest, numValues, srcDataType, destDataType)
void* src;           // Input data in format specified by srcDataType
void* dest;          // Output data in format specified by destDataType
long numValues;      // # of values to convert
int srcDataType;     // Igor code for input data format
int destDataType;    // Igor code for output data format
```

This routine calls the appropriate conversion routine based on the srcDataType and destDataType parameters.

ConvertData2 can handle any combination of the legal Igor numeric data types. It takes the description of source and destination data types in the form Igor number type codes, defined in IgorXOP.h. Both the srcDataType and destDataType can be any of the following:

```
NT_FP64                    // Double-precision IEEE float
NT_FP32                    // Single-precision IEEE float
NT_I32                     // 32 bit signed integer
NT_I16                     // 16 bit signed integer
NT_I8                      // 8 bit signed integer
NT_FP32 | NT_UNSIGNED      // 32 bit unsigned integer
NT_I16 | NT_UNSIGNED       // 16 bit unsigned integer
NT_I8 | NT_UNSIGNED        // 8 bit unsigned integer
```

The number type should *not* include NT_CMPLX. If the data is complex, the numValues parameter should reflect that.

Returns zero if everything is OK, 1 if the conversion is not supported (e.g., converting to 2 byte floating point), -1 if no conversion is needed (source format == dest format).

```
int
NumTypeToNumBytesAndFormat(numType, numBytesPerValuePtr, dataFormatPtr,
   isComplexPtr)
int numType;               // Igor number type code.
int* numBytesPerValuePtr;  // Output: number of bytes per value.
int* dataFormatPtr;        // Output: XOP Toolkit data format code.
int* isComplexPtr;         // Output: True if complex.
```

This routine converts standard Igor number type codes (e.g., NT_FP64), which are defined in IgorXOP.h, into the XOP Toolkit number format codes (IEEE_FLOAT, SIGNED_INT, and UNSIGNED_INT), defined in XOPSupport.h.

The value returned via numBytesPerValuePtr is the number of bytes per value, not the number of bytes per point. In other words, if a point of a wave is complex, it will take twice as many bytes as returned via numBytesPerValuePtr .

```
int
NumBytesAndFormatToNumType(numBytesPerValue, dataFormat, numTypePtr)
int numBytesPerValue;       // Number of bytes per value.
int dataFormat;             // XOP Toolkit data format code.
int* numTypePtr;            // Output: Igor number type code.
```

This routine is converts XOP Toolkit data format codes (IEEE_FLOAT, SIGNED_INT, and UNSIGNED_INT), defined in XOPSupport.h, into standard Igor number type codes (e.g., NT_FP64), defined in IgorXOP.h.

```
void
ScaleData(dataType, dataPtr, offsetPtr, multiplierPtr, numValues)
int dataType;               // Code for input data format
void* dataPtr;              // Pointer to data to be scaled
double* offsetPtr;          // Pointer to the offset value
double* multiplierPtr;      // Pointer to the multiplier value
long numValues;             // Code for output data format
```

Scales the data pointed to by dataPtr by adding the offset and multiplying by the multiplier.

dataType is one of the Igor numeric type codes defined in IgorXOP.h (same as for ConvertData2).

The NT_CMPLX bit must *not* be set. If the data is complex, this must be reflected in the numValues parameter.

ScaleData relies on a number of low-level scaling routines with names of the form Scale<Type> where <Type> can be any of the following:

Double, Float, Long, Short, Byte, UnsignedLong, UnsignedShort, UnsignedByte

For most uses, you should use the high level ScaleData routine.

```
void
ScaleClipAndRoundData(dataType, dataPtr, numValues, offset, multiplier,
                                                dMin, dMax, doRound)
int dataType;               // Code for input data format
void* dataPtr;              // Pointer to data to be scaled
long numValues;             // Code for output data format
double offset;              // Offset value
double multiplier;          // Multiplier value
double dMin;                // Min value for clipping
double dMax;                // Max value for clipping
int doRound;                // If non-zero, rounding is performed
```

Scales the data pointed to by dataPtr by adding the offset and multiplying by the multiplier. If offset is 0.0 and multiplier is 1.0, no scaling is done.

Clips to the specified min and max. If min is -INF and max is +INF, no clipping is done. If min and max are both zero, integer data is clipped to the minimum and maximum value that can be represented by the data type.

If doRound is non-zero, the data is rounded to the nearest integer.

All calculations are done in double precision.

dataType is one of the Igor numeric type codes defined in IgorXOP.h (same as for ConvertData2).

The NT_CMPLX bit must *not* be set. If the data is complex, this must be reflected in the numValues parameter.

```
void
FixByteOrder(p, bytesPerPoint, numValues)
void* p;                    // Pointer to input data of any type
int bytesPerPoint;          // Bytes in one point of input format
long numValues;             // Number of values to fix
```

The routine converts data from Motorola (high byte first) to Intel (low byte first) format or vice-versa.

# Routines for Dealing With Object Names

If your XOP creates a new data object (a wave, variable or data folder) you need to provide a legal name. The name must also be unique, unless you are overwriting an existing object. The CreateValidObjectName routine, described below, is appropriate for most uses and does all necessary legality and conflict checking.

If you pass a string to Igor for execution as a command, using FinishDialogCmd, XOPCommand or XOPSilentCommand, you must quote liberal object names. Use PossiblyQuoteName and CatPossiblyQuotedName for this purpose. See the Igor Pro manual for general information on liberal names.

```
int
UniqueName(baseName, finalName)
char* baseName;            // C string containing base name
char* finalName;           // C string containing new, unique name
```

New XOPs should use CreateValidDataObjectName or UniqueName2 instead of UniqueName.

UniqueName generates a name that does not conflict with an existing name.

This is used by operations that want to create waves and auto-name them (like LoadWaves/A).

The finalName is derived by appending one or more digits to the baseName. finalName must be able to hold MAX_OBJ_NAME+1 bytes.

Make sure that the baseName is not too long. Otherwise, the finalName could be too long to be a legal name.

The function result is the number that was appended to the baseName to make the finalName.

```
int
UniqueName2(nameSpaceCode, baseName, finalName, suffixNumPtr)
int nameSpaceCode;         // Usually MAIN_NAME_SPACE. See IgorXOP.h
char* baseName;            // Base name to use for creating unique name
char* finalName;           // Receives unique name
long suffixNumPtr;         // Keeps track of last suffix used
```

If your goal is to generate a valid object name that you can use to create a data object, consider using CreateValidDataObjectName instead of this routine.

UniqueName2 generates a name that does not conflict with an existing name.

The function result is 0 if OK, -1 for a bad nameSpaceCode or some other error code.

This is an improved version of UniqueName. Given a base name (like "wave") UniqueName2 returns a name (like "wave3") via finalName that does not conflict with any existing names. finalName must be able to hold MAX_OBJ_NAME+1 bytes.

*suffixNumPtr is both an input and an output. Its purpose is to speed up the search for unique names when creating a batch of names in a loop. The number appended to make the name unique will be *suffixNumPtr or greater. Igor sets *suffixNumPtr to the number Igor used to make the name unique. Typically, you should set *suffixNumPtr to zero before your first call to UniqueName2.

nameSpaceCode is MAIN_NAME_SPACE for Igor's main name space (waves, variables, windows).

nameSpaceCode is DATA_FOLDER_NAME_SPACE for data folders.

See IgorXOP.h for other less frequently-used name space codes.

```
int
SanitizeWaveName(waveName, column)
char* waveName;       // Input and output wave name
long column;          // Number of column being loaded or zero
```

New XOPs should use the CleanupName, CheckName and CreateValidDataObjectName routines instead of SanitizeWaveName.

SanitizeWaveName is intended mainly for use in file-loader XOPs such as SimpleLoadWave but may have other uses.

Given a pointer to a C string containing a proposed wave name, SanitizeWaveName changes it to make it a valid wave name if necessary. It returns 1 if it had to make a change, 0 if name was OK to begin with. waveName must be able to hold MAX_OBJ_NAME+1 bytes.

First SanitizeWaveName truncates the proposed name if it is too long. Then it makes sure that the first character is alphabetic. Then it replaces any subsequent characters that are not alphanumeric with underscore.

column should be the number of the column being loaded or the number of the wave in a set of waves or zero if this does not apply to your XOP.

```
int
CleanupName(beLiberal, name, maxNameChars)
int beLiberal;
char* name;
int maxNameChars;
```

If your goal is to generate a valid object name that you can use to create a data object, consider using CreateValidDataObjectName instead of this routine.

CleanupName changes the name, if necessary, to make it a legal Igor object name.

For most uses, pass MAX_OBJ_NAME for the maxNameChars parameter. name must be able to hold maxNameChars+1 bytes.

Igor Pro 3.0 and later allows wave and data folder names to contain characters, such as space and dot, that were previously illegal in names. We call this “liberal” name rules.

If beLiberal is non-zero, CleanupName uses liberal name rules. Liberal rules are allowed for wave and data folder names but are not allowed for string and numeric variable names so pass zero for beLiberal for these objects.

If you are going to use the name in Igor’s command line (via the Execute operation or via the XOPCommand or XOPSilentCommand callbacks), and if the name uses liberal rules, the name needs to be single-quoted. In these cases, you should call PossiblyQuoteName after calling CleanupName.

```
int
CheckName(dataFolderH, objectType, name)
DataFolderHandle dataFolderH;
int objectType;
char name[MAX_OBJ_NAME+1];
```

If your goal is to generate a valid object name that you can use to create a data object, consider using CreateValidDataObjectName instead of this routine.

CheckName checks the name for legality and uniqueness.

If dataFolderH is NULL, it looks for conflicts with objects in the current data folder. If it is not NULL, it looks for conflicts in the folder specified by dataFolderH.

objectType is one of the following (defined in IgorXOP.h):

```
WAVE_OBJECT, VAR_OBJECT (numeric variable), STR_OBJECT (string variable)
GRAPH_OBJECT, TABLE_OBJECT, LAYOUT_OBJECT
PANEL_OBJECT, NOTEBOOK_OBJECT
DATAFOLDER_OBJECT, PATH_OBJECT, PICT_OBJECT
```

The function result is 0 if the name is legal and is not in conflict with an existing object.

Returns an Igor error code otherwise.

```
int
CreateValidDataObjectName(
DataFolderHandle dataFolderH,     // Handle to data folder or NULL
char* inName,                     // Input: Proposed name
char* outName,                    // Output: Valid name
long* suffixNumPtr,               // Used to make name unique
int objectType,                   // Type code from IgorXOP.h
int beLiberal,                    // 1 to allow liberal names
int allowOverwrite,               // 1 if it is OK to overwrite object
int inNameIsBaseName,             // 1 if inName needs digits appended
int printMessage,                 // 1 to print message about conflict
int* nameChangedPtr,              // Output: 1 if name changed
int* doOverwritePtr)              // Output: 1 if need to overwrite object
```

This routine is designed to do all of the nasty work needed to get a legal name for a given object. It cleans up illegal names and resolves name conflicts.

It returns in outName a name that can be safely used to create a data object of a given type (wave, string variable, numeric variable, data folder).

inName is the proposed name for the object or a base name to which one or more digits is to be added.

outName is the name after possible cleanup and uniquification. outName must be able to hold MAX_OBJ_NAME+1 bytes.

*suffixNumPtr is both an input and an output. Its purpose is to speed up the search for unique names when creating a batch of names in a loop. The number appended to make the name unique will be *suffixNumPtr or greater. Igor sets *suffixNumPtr to the number Igor used to make the name unique. Typically, you should set *suffixNumPtr to zero before your first call to CreateValidDataObject.

dataFolderH is a handle to a data folder or NULL to use the current data folder.

objectType is one of the following:

WAVE_OBJECT, VAR_OBJECT, STR_OBJECT, DATAFOLDER_OBJECT

beLiberal is 1 if you want to allow liberal names or 0 if not. If the object type is VAR_OBJECT or STR_OBJECT, the name will not be liberal, even if beLiberal is 1. Igor allows only wave and data folder names to be liberal.

allowOverwrite is 1 if it is OK for outName to be the name of an existing object of the same type.

inNameIsBaseName should be 1 if inName is a base name (e.g., "wave") to which a suffix (e.g., "0") must always be added to produce the actual name (e.g., "wave0"). If inNameIsBaseName is 0 then no suffix will be added to inName unless it is needed to make the name unique.

printMessage is 1 if you want CreateValidDataObjectName to print a message in Igor's history area if an unexpected name conflict occurs. A message is printed if you are not using a base name and not allowing overwriting and there is a name conflict. A message is also printed if a conflict with an object of a different type prevents the normal name from being used.

CreateValidDataObjectName sets *nameChangedPtr to the truth that outName is different from inName.

It sets *doOverwritePtr to the truth that outName is the name of an existing object of the same type and allowOverwrite is 1.

inName and outName can point to the same array if you don't want to preserve the original name. Both must be big enough to hold MAX_OBJ_NAME+1 bytes.

The function result is 0 if OK or an error code.

***Example***

This is a simplified section of code from GBLoadWave which uses CreateValidDataObjectName. lpp is a pointer to a structure containing parameters for the load operation. ciHandle is a locked handle containing an array of records, one for each wave to be created.

```
long column, suffixNum;
char base[MAX_OBJ_NAME+1];
int nameChanged, doOverwrite;
int result;

strcpy(base, "wave");          // Base name for waves.
suffixNum = 0;
for (column = 0; column < lpp->numArrays; column++) {
   ciPtr = *ciHandle + column;
   ciPtr->points = lpp->arrayPoints;
   strcpy(ciPtr->waveName, base);

   // Take care of illegal or conflicting names.
   result = CreateValidDataObjectName(NULL, ciPtr->waveName,
               ciPtr->waveName, &suffixNum, WAVE_OBJECT, 1,
               lpp->flags & OVERWRITE, 1, 1, &nameChanged, &doOverwrite);

   ciPtr->wavePreExisted = doOverwrite;

   if (result == 0)
      result = MakeAWave(ciPtr->waveName, lpp->flags, &ciPtr->waveHandle,
                           lpp->arrayPoints, lpp->outputDataType);

   if (result) {      // Couldn't make wave (probably low memory) ?
      <Clean up>;
      return result;
   }
}
```

```
int
PossiblyQuoteName(name)
char name[MAX_OBJ_NAME+2+1];
```

PossiblyQuoteName puts single quotes around the Igor object name if they would be needed to use the name in Igor's command line.

Igor Pro 3.0 and later allows wave and data folder names to contain characters, such as space and dot, that were previously illegal in names. We call this "liberal" name rules.

If an object has such a name, you must single-quote the name to use it in Igor's command line. This includes using it in the Execute operation or in the XOPCommand, XOPSilentCommand, or FinishDialogCmd XOPSupport routines. Thus, if you are going to use a wave or data folder name for this purpose, you should call PossiblyQuoteName to add the quotes if needed.

**NOTE**: name must be able to hold two additional bytes (MAX_OBJ_NAME+2 plus the terminating null).

**NOTE**: Liberal rules are not allowed for string and numeric variable names.

PossiblyQuoteName returns 0 if the name is not liberal, 1 if it is liberal.

```
int
CatPossiblyQuotedName(str, name)
char* str;
char name[MAX_OBJ_NAME+2+1];
```

Adds the specified Igor object name to the end of the string. If necessary, puts single quotes around the name so that it can be used in the Igor command line.

Use this to concatenate a wave name to the end of a command string when the wave name may be a liberal name that needs to be quoted to be used in the command line.

See **PossiblyQuoteName** for details on liberal names.

Remember that the quoted name will take an extra two bytes. str must be big enough to hold it.

CatPossiblyQuotedName returns 0 if the name is not liberal, 1 if it is liberal.

***Example***

```
char waveName[MAX_OBJ_NAME+1];          // This contains a wave name.
char cmd[256];
strcpy(cmd, "Display ");
CatPossiblyQuotedName(cmd, waveName);
XOPSilentCommand(cmd);
```

# Color Table Routines

These routines were added to the XOP Toolkit for use by advanced programmers who want to present a user interface with the same color tables as Igor itself presents. An example is the Surface Plotter XOP which allows the user to choose from a set of Igor color tables.

The Surface Plotter calls GetIndexedIgorColorTableName to display a list of Igor color tables in a popup menu. When it needs to know what colors are in a table, it calls GetNamedIgorColor-TableHandle to get a color table handle, GetIgorColorTableInfo to find the number of colors in the table, and GetIgorColorTableValues to find the actual colors.

```
int
GetIndexedIgorColorTableName(index, name)
int index;                  // Zero-based index of a color table
char name[MAX_OBJ_NAME+1]; // Output: name of that color table
```

Returns via name the name of a color table indicated by the index or "" if the index is invalid.

Valid indices start from zero. You can find the maximum valid index by calling this routine with increasing indices until it returns an error.

The function result is 0 if OK or a non-zero error code if the index is invalid.

```
int
GetNamedIgorColorTableHandle(name, ictHPtr)
const char* name;                    // Name of a color table
IgorColorTableHandle* ictHPtr;       // Handle to color table info
```

Returns via *ictHPtr a handle to an Igor color table or NULL in case of error.

The returned handle belongs to Igor. Do not modify or delete it.

The name parameter is case insensitive.

Igor Pro 4 contained the following color tables: Rainbow, Grays, YellowHot, BlueHot, BlueRedGreen, RedWhiteBlue, PlanetEarth, Terrain. Igor Pro 5 added many more color tables. You can find the names of all color tables using GetIndexedIgorColorTableName.

The function result is 0 if OK or a non-zero error code.

```
int
GetIgorColorTableInfo(ictH, name, numColorsPtr)
IgorColorTableHandle ictH; // Handle from GetNamedIgorColorTableHandle
char name[MAX_OBJ_NAME+1]; // Output: name of color table
int* numColorsPtr;         // Output: number of colors in table
```

Provides access to the name and the number of colors in the Igor color table specified by ictH.

ictH is a handle that you got by calling GetNamedIgorColorTableHandle.

If you don't want to know the name, pass NULL for name. If you don't want to know the number of colors, pass NULL for numColorsPtr.

The function result is 0 if OK or a non-zero error code.

```
int
GetIgorColorTableValues(ictH, startIndex, endIndex, updateVals, csPtr)
IgorColorTableHandle ictH; // Handle from GetNamedIgorColorTableHandle
int startIndex;            // Index of first color of interest
int endIndex;              // Index of last color of interest
int updateVals;
IgorColorSpec* csPtr;      // Output: Color values go here
```

Returns via csPtr a description of the colors associated with the Igor color table specified by ictH.

ictH is a handle that you got by calling GetNamedIgorColorTableHandle.

startIndex and endIndex specify the indices of the colors for which you want to get a description. startIndex must be between 0 and the number of colors in the table minus one. endIndex must be between startIndex and the number of colors in the table minus one. You can find the number of colors in the table using GetIgorColorTableInfo.

The IgorColorSpec structure contains an RGBColor field which identifies the RGB color for a color table entry with a particular index. These structures are defined in IgorXOP.h.

The value field of the IgorColorSpec structure tells you the pixel value that would need to be written to video RAM to get the associated color to appear on the screen when the monitor is in 16 or 256 color mode. It is typically used by advanced programmers who are writing directly to offscreen bitmap memory.

However, when a monitor is running in 16 or 256 color mode, this value is invalidated whenever the system changes the hardware color lookup table, which can happen at any time. If you pass non-zero for the updateVals parameter, then Igor will update the value field for each color before returning it to you and it will be accurate until the next time the system changes the hardware color lookup table. If you pass zero for the updateVals parameter, then Igor will not update the value field and it is likely to be stale.

Updating the value fields takes time so you should pass non-zero for the updateVals parameter only if you really need accurate pixel values. For example, if you just want to know what RGB

colors appear in a particular color table then you don't need the pixel values and should pass 0 for the updateVals parameter. On the other hand, if you are writing into an offscreen bitmap in preparation for blasting it to the screen, then you need accurate pixel values and you should pass 1 for updateVals.

The function result is 0 if OK or a non-zero error code.

# Routines for Dealing With Igor Procedures

These routines allow an XOP to get and set procedures in the Igor procedure window. Most XOPs will not need these routines.

```
int
GetIgorProcedureList(Handle* hPtr, long flags)
Handle* hPtr;
long flags;
```

The main use for this routine is to check if a particular macro or function exists in the Igor procedure windows.

GetIgorProcedureList returns via *hPtr a handle to a semicolon-separated list of procedure names. Depending on the flags parameter, the list may contain names of macros, names of user-defined functions, or names of both macros and user-defined functions.

Note that the text in the handle is *not* null terminated. Use GetHandleSize to find the number of bytes in the handle. To use C string functions on this text you need to copy it to a local buffer and null-terminate it or add a null terminator to the handle and lock the handle. See **Understand the Difference Between a String in a Handle and a C String** on page 321.

This handle belongs to you, so call DisposeHandle to dispose it when you no longer need it.

If Igor can not build the list, it returns a non-zero error code and sets *hPtr to NULL.

The flags parameter is defined as follows:

| | |
|---|---|
| Bit 0 | If set, GetIgorProcedureList will list all macros. If cleared, it will ignore all macros. |
| Bit 1 | If set, GetIgorProcedure will list all user-defined functions. If cleared, it will ignore all user-defined functions. |
| All other bits | Reserved for future use and must be set to 0. |

Igor will be unable to build the list if a syntactical error in the procedure files prevents Igor from successfully scanning them. In this case, GetIgorProcedureList will return NEED_COMPILE.

GetIgorProcedureList can also return NOMEM if it runs out of memory. This is unlikely.

Future versions of GetIgorProcedureList may return other error codes so your XOP should not crash or otherwise grossly misbehave if it receives some other error code.

```
int
GetIgorProcedure(procedureName, hPtr, flags)
const char* procedureName;
Handle* hPtr;
long flags;
```

The main use for this routine is to check if a particular macro or function exists in the Igor procedure windows.

If Igor can find the procedure (macro or function) specified by procedureName, GetIgor-Procedure returns via *hPtr a handle to the text for the procedure and returns a result of 0. The handle will contain the text for the specified procedure with a carriage return at the end of each line.

Note that the text in the handle is *not* null terminated. Use GetHandleSize to find the number of bytes in the handle. To use C string functions on this text you need to copy it to a local buffer and null-terminate it or add a null terminator to the handle and lock the handle. See **Understand the Difference Between a String in a Handle and a C String** on page 321. If you pass the handle back to Igor, you must remove the null terminator and unlock the handle.

This handle belongs to you, so call DisposeHandle to dispose it when you no longer need it.

If Igor can not find the procedure, it returns a non-zero error code and sets *hPtr to NULL.

The flags parameter is defined as follows:

| | |
|---|---|
| Bit 0 | If set, GetIgorProcedure will look for macros with the specified name. If cleared, it will ignore all macros. |
| Bit 1 | If set, GetIgorProcedure will look for user-defined functions with the specified name. If cleared, it will ignore all user-defined functions. |
| All other bits | Reserved for future use and must be set to 0. |

Igor will be unable to find the procedure if there is no such procedure. In this case, GetIgorProcedure will return NO_MACRO_OR_FUNCTION.

Igor will be unable to find the procedure if a syntactical error in the procedure files prevents Igor from successfully scanning them. In this case, GetIgorProcedure will return NEED_COMPILE.

GetIgorProcedure can also return NOMEM if it runs out of memory. This is unlikely.

Future versions of GetIgorProcedure may return other error codes so your XOP should not crash or otherwise grossly misbehave if it receives some other error code.

```
int
SetIgorProcedure(procedureName, h, flags)
const char* procedureName;
Handle h;
long flags;
```

This routine is used by very advanced XOPs, like the Surface Plotter, that add a target window type to Igor.

The handle h belongs to Igor. Once you pass it to SetIgorProcedure, you must not modify it, access it, or dispose of it.

See **Adding XOP Target Windows** on page 257 for further information.

```
enum CloseWinAction
DoWindowRecreationDialog(procedureName)
char* procedureName;
```

This routine is used by very advanced XOPs, like the Surface Plotter, that add a target window type to Igor. See **Adding XOP Target Windows** on page 257 for further information.

```
int
GetFunctionInfo(name, fip)
const char* name;
FunctionInfoPtr fip;
```

Returns information that you need in order to call an Igor user function or an external function from an XOP. You might want to do this, for example, to implement your own user-defined curve fitting algorithm. This is an advanced feature that most XOP programmers will not need.

For an overview see **Calling User-Defined and External Functions** on page 285.

name is the name of an existing user or external function. If there is no such function, GetFunctionInfo returns an error. If the function is a user function and procedures are not in a compiled state, GetFunctionInfo returns an error. If everything is OK, GetFunctionInfo returns zero.

The information returned by GetFunctionInfo should be used and then discarded. If the user does anything to cause procedures to be compiled then the values returned by GetFunctionInfo are no longer valid.

GetFunctionInfo returns via fip->compilationIndex a value that you will pass to CallFunction. This value is used to make sure that procedures have not been changed between the time you call GetFunctionInfo and the time you call CallFunction.

GetFunctionInfo returns via fip->functionID a value which you will pass to CallFunction to specify which user function you want to execute.

GetFunctionInfo returns via fip->subType a code that identifies certain special purpose functions. The value returned currently has no use but may be used in the future.

GetFunctionInfo returns via fip->isExternalFunction a value that is non-zero for external functions and zero for user functions. This field is for your information only. Your code will be the same for external and user functions.

GetFunctionInfo returns via fip->returnType one of the following codes:

```
NT_FP64:              Return value is a double-precision number
NT_FP64 | NT_CMPLX:   Return value is a complex double-precision number
HSTRING_TYPE:         Return value is a string
```

GetFunctionInfo returns via fip->numOptionalParameters, fip->numRequiredParameters and fip->totalNumParameters the number of optional, required and total parameters for the function. Currently, an XOP can call a user function that has optional parameters but the XOP can not pass optional parameters to the function. In other words, it must pass the required parameters only.

GetFunctionInfo returns via fip->parameterTypes an array of parameter types. GetFunctionInfo stores a parameter type value in elements 0 through fip->totalNumParameters-1. Elements fip->totalNumParameters and higher are undefined so you must not use them.

You must use the CheckFunctionForm XOPSupport routine to make sure that the function is of the form you want. You normally don't need to examine the parameter type values directly, but in case you are curious, see the comments for GetFunctionInfo in XOPSupport.c.

Parameter type values for numeric, complex numeric and string parameters may be ORed with FV_REF_TYPE. This indicates that the corresponding parameter is "pass-by-reference", meaning that the function can change the value of that parameter.

Added for Igor Pro 5.00. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

```
int
GetFunctionInfoFromFuncRef(fRef, fip)
FUNCREF fRef;
FunctionInfoPtr fip;
```

Returns information that you need in order to call an Igor user function or an external function from an XOP. You might want to do this, for example, to implement your own user-defined curve fitting algorithm. This is an advanced feature that most XOP programmers will not need.

For an overview see **Calling User-Defined and External Functions** on page 285.

fRef is the value of a FUNCREF field in an Igor Pro structure passed to your XOP as a parameter.

GetFunctionInfoFromFuncRef will return an error if the function is a user function and procedures are not in a compiled state. If everything is OK, GetFunctionInfoFromFuncRef returns zero.

GetFunctionInfoFromFuncRef works just like GetFunctionInfo except that you pass in a FUNCREF instead of the name of the function. See the documentation for GetFunctionInfo for further details.

Added for Igor Pro 5.03. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

```
int
CheckFunctionForm(fip, requiredNumParameters, requiredParameterTypes,
                                    badParameterNumberPtr, returnType)
FunctionInfoPtr fip;
int requiredNumParameters;
int requiredParameterTypes[];
int* badParameterNumberPtr;
int returnType;
```

Checks the form (number of parameters, types of parameters and return type) of an Igor user-defined or external function against the required form. This is an advanced feature that most XOP programmers will not need.

For an overview see **Calling User-Defined and External Functions** on page 285.

You must call CheckFunctionForm before calling CallFunction to make sure that the function you are calling has the form you expect. Otherwise you may cause a crash.

fip is pointer to a structure set by calling GetFunctionInfo.

requiredNumParameters is the number of parameters you expect the function to have.

requiredParameterTypes is an array in which you have set each value to one of the following:

```
NT_FP64                       The parameter must be scalar numeric
NT_FP64 | NT_CMPLX            The parameter must be complex numeric
HSTRING_TYPE                  The parameter must be a string
WAVE_TYPE                     The parameter must be a scalar numeric wave
WAVE_TYPE | NT_CMPLX          The parameter must be a complex numeric wave
TEXT_WAVE_TYPE                The parameter must be a text wave
FV_FUNC_TYPE                  The parameter must be a function reference
FV_STRUCT_TYPE | FV_REF_TYPE  The parameter must be a structure
```

The number of elements in requiredParameterTypes must be at least equal to requiredNumParameters.

If the parameter must be pass-by-reference, use FV_REF_TYPE in addition to the values above. For example:

```
NT_FP64 | FV_REF_TYPE
NT_FP64 | NT_CMPLX | FV_REF_TYPE
HSTRING_TYPE | FV_REF_TYPE
FV_STRUCT_TYPE | FV_REF_TYPE
```

Pass-by-reference is applicable to numeric, string and structure parameters only. Numeric and string parameters can be passed by value or passed by reference but structure parameters are always passed by reference.

If you do not want CheckFunctionForm to check a particular parameter, pass -1 in the corresponding element of the requiredParameterTypes array.

Calling a function with at structure parameter requires Igor Pro 5.04 or later. For background information see **Structure Parameters** on page 168.

When dealing with a structure parameter, CheckFunctionForm can not guarantee that your XOP and the function you are calling are using the same definition of the structure. See **Structure Parameters** on page 168 for some suggestions for dealing with this problem.

If the function is an external function which takes a wave parameter, there is no way to know if the external function expects a numeric wave or a text wave. Consequently, CheckFunctionForm does not distinguish between numeric and text waves for external functions. The external function itself is supposed to check the type of the wave passed in at runtime and return an error if it is the wrong type.

returnType is the required return type of the function which must be one of the following:

```
NT_FP64
NT_FP64 | NT_CMPLX
HSTRING_TYPE
```

If you do not want CheckFunctionForm to check the return type, pass -1 as the returnType parameter.

CheckFunctionForm sets *badParameterNumberPtr to the zero-based index of the first parameter that does not match the required type or to -1 if all parameters match the required type.

It returns 0 if the form matches the requirements or an error code if not.

If a function parameter type does not match the required parameter type, the error code returned will indicate the type of parameter required but not which parameter type was bad. If you want to inform the user more specifically, use the value returned via badParameterNumberPtr to select your own more specific error code. If the error was a parameter type mismatch, *badParameterNumberPtr will contain the zero-based index of the bad parameter. Otherwise, *badParameterNumberPtr will contain -1.

Added for Igor Pro 5.00. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

```
int
CallFunction(fip, parameters, resultPtr)
FunctionInfoPtr fip;
void* parameters;
void* resultPtr;
```

Calls the Igor user-defined or external function identified by fip. This is an advanced feature that most XOP programmers will not need.

For an overview see **Calling User-Defined and External Functions** on page 285.

fip is a pointer to a FunctionInfo structure whose values were set by calling GetFunctionInfo.

fip->compilationIndex is used by CallFunction to make sure that procedures were not recompiled after you called GetFunctionInfo. If procedures were recompiled, then the information in the structure may be invalid so CallFunction returns BAD_COMPILATION_INDEX.

parameters is a pointer to a structure containing the values that you want to pass to the function. These values must agree in type with the function's parameter list, as indicated by the parameter information that you obtain via GetFunctionInfo. To guarantee this, you must call CheckFunctionForm before calling CallFunction.

**NOTE**: The parameters structure must use standard XOP structure packing, namely, two-byte packing. If you don't set the structure packing correctly, a crash is likely. See **Structure Alignment** on page 279.

Parameter type values are discussed in detail in the comments for the GetFunctionInfo function in XOPSupport.c. Here is the basic correspondence between function parameter types and the respective structure field:

```
if (parameterType == NT_FP64)
   structure field is double

if (parameterType == (NT_FP64 | NT_CMPLX))
   structure field is double[2]

if (parameterType == HSTRING_TYPE)
   structure field is Handle

if (WAVE_TYPE bit is set)
   structure field is waveHndl

if (FV_FUNC_TYPE bit is set)
   structure field is long
```

**NOTE**: For pass-by-value strings parameters, ownership of a handle stored in the parameter structure is passed to the function when you call CallFunction. The function will dispose the

handle and CallFunction will set the field to NULL. You must not dispose it or otherwise reference it after calling CallFunction.

**NOTE**: For pass-by-reference string parameters, the handle stored in the parameter structure field may be reused or disposed by the called function. When CallFunction returns, you own the handle which may be the same handle you passed or a different handle. You own this handle and you must dispose it when you no longer need it. If the field is NULL, which could occur in the event of an error, you must not dispose of it or otherwise access it.

CallFunction stores the function result at the location indicated by resultPtr. Here is the correspondence between the function result type and the variable pointed to by resultPtr:

```
NT_FP64              double
NT_FP64 | NT_CMPLX   double[2]
HSTRING_TYPE         Handle
```

**NOTE**: A function that returns a string can return NULL instead of a valid handle. You must test the returned value to make sure it is not NULL before using it. If the returned handle is not NULL, you own it and you must dispose it when you no longer need it.

CallFunction returns 0 as the function result if the function executed. If the function could not be executed, it returns a non-zero error code.

Added for Igor Pro 5.00. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

# Windows-Specific Routines

These routines are available and applicable to XOPs running under the Windows OS only.

```
HMODULE
XOPModule(void)
```

This routine is supported on Windows only.

XOPModule returns the XOP's module handle.

You will need this HMODULE if your XOP needs to get resources from its own executable file using the Win32 FindResource and LoadResource routines. It is also needed for other Win32 API routines.

```
HMODULE
IgorModule(void)
```

This routine is supported on Windows only.

IgorModule returns Igor's module handle.

There is probably no reason for an XOP to call this routine.

```
HWND
IgorClientHWND(void)
```

This routine is supported on Windows only.

IgorClientHWND returns Igor's MDI client window HWND.

Some Win32 API calls require that you pass an HWND to identify the owner of a new window or dialog. An example is MessageBox. You must pass IgorClientHWND() for this purpose.

```
int
WMGetLastError(void)
```

This routine is supported on Windows only.

WMGetLastError does the same as the Win32 GetLastError routine except for three things. First, it translates Windows OS error codes into codes that mean something to Igor. Second, it always returns a non-zero result whereas GetLastError can sometimes return 0. Third, it calls SetLastError(0).

For a detailed explanation, see **Handling Windows OS Error Codes** on page 128.

```
int
WindowsErrorToIgorError(int winErr)
int winErr;           // Windows OS error code
```

This routine is supported on Windows only.

WindowsErrorToIgorError takes a Windows OS error code and returns an error code that means something to Igor.

For a detailed explanation, see **Handling Windows OS Error Codes** on page 128.

```
int
SendWinMessageToIgor(hwnd, iMsg, wParam, lParam, beforeOrAfter)
HWND hwnd;           // Your XOP's HWND
UINT iMsg;           // The message your window procedure received
WPARAM wParam;       // The wParam that came with the message
LPARAM lParam;       // The lParam that came with the message
int beforeOrAfter;   // 0 if you before you service message, 1 if after
```

If your XOP adds a window to Igor, you must call SendWinMessageToIgor twice from your window procedure - once before you process the message and once after. You must do this for every message that you receive.

Calling SendWinMessageToIgor allows Igor to do certain housekeeping operations that are needed so that your window will fit cleanly into the Igor environment.

If the function result from SendWinMessageToIgor is non-zero, you should skip processing of the message. For example, Igor returns non-zero for click and key-related messages while an Igor procedure is running.

To help you understand why this is necessary, here is a description of what Igor does with these messages as of this writing.

**NOTE**: Future versions of Igor may behave differently, so you must send every message to Igor, once before you process it and once after.

| Message | Action |
|---|---|
| WM_CREATE | Before: Allocates memory used so that the XOP window appears in the Windows menu and can respond to user actions like Ctrl-E (send behind) and Ctrl-W (close).<br><br>After: Nothing. |
| WM_DESTROY | Before: Nothing.<br><br>After: Deallocates memory allocated by WM_CREATE. |
| WM_MDIACTIVATE<br><br>(when XOP window is being activated only) | Before: Compiles procedure windows if necessary.<br><br>After: Sets Igor menu bar (e.g., removes "Graph" menu from Igor menu bar). |

See **Adding a Simple Window on Windows** on page 251 for background information.

# Miscellaneous Routines

These routines don't fit in any particular category.

```
int
XOPCommand(cmdPtr)
char* cmdPtr;                // C string containing an Igor command
```

Submits a command to Igor for execution.

The function result is 0 if OK or an error code.

Use this to access Igor features for which there is no direct XOP interface.

cmdPtr is a C string (null byte at the end). The string must consist of one line of text not longer than MAXCMDLEN and with no carriage return characters.

A side-effect of XOPCommand is that it causes Igor to do an update. See DoUpdate below for a definition of "update".

XOPCommand displays the command being executed in Igor's command line. For most cases, use XOPSilentCommand, which does not display the command, instead of XOPCommand.

Liberal names of waves and data folders must be quoted before using them in the Igor command line. Use PossiblyQuoteName when preparing the command to be executed so that your XOP works with liberal names.

**NOTE**: If your XOP adds a window to Igor or if the command that you are executing directly or indirectly calls your XOP again, this callback will cause recursion. This can cause your XOP to crash if you do not handle it properly. See **Handling Recursion** on page 137 for details.

```
int
XOPCommand2(cmdPtr, silent, sendToHistory)
char* cmdPtr;                // C string containing an Igor command
int silent;                  // True to not show cmd in command line
int  sendToHistory;          // True to send cmd to history area
```

Submits a command to Igor for execution.

The function result is 0 if OK or an error code.

Use this to access Igor features for which there is no direct XOP interface.

cmdPtr is a C string (null byte at the end). The string must consist of one line of text not longer than MAXCMDLEN and with no carriage return characters.

If silent is non-zero, the command is displayed in Igor's command line while it executes.

If sentToHistory is non-zero and the command generates no error, the command is sent to the history area after execution.

A side-effect of XOPCommand is that it causes Igor to do an update. See DoUpdate below for a definition of “update”.

XOPCommand displays the command being executed in Igor’s command line. For most cases, use XOPSilentCommand, which does not display the command, instead of XOPCommand.

Liberal names of waves and data folders must be quoted before using them in the Igor command line. Use PossiblyQuoteName when preparing the command to be executed so that your XOP works with liberal names.

**NOTE**: If your XOP adds a window to Igor or if the command that you are executing directly or indirectly calls your XOP again, this callback will cause recursion. This can cause your XOP to crash if you do not handle it properly. See **Handling Recursion** on page 137 for details.

```
int
XOPSilentCommand(cmdPtr)
char* cmdPtr;              // C string containing an Igor command
```

Submits a command to Igor for execution.

The function result is 0 if OK or an error code.

Use this to access Igor features for which there is no direct XOP interface.

A side-effect of XOPSilentCommand is that it causes Igor to do an update. See DoUpdate below for a definition of “update”.

This is just like XOPCommand except that it does not flash the command being executed in Igor’s command line. Use this if your XOP needs to execute an Igor command at a time when the user does not expect to see things flashing, such as during IDLE processing.

Liberal names of waves and data folders must be quoted before using them in the Igor command line. Use PossiblyQuoteName when preparing the command to be executed so that your XOP works with liberal names.

**NOTE**: If your XOP adds a window to Igor or if the command that you are executing directly or indirectly calls your XOP again, this callback will cause recursion. This can cause your XOP to crash if you do not handle it properly. See **Handling Recursion** on page 137 for details.

```
void
DoUpdate(void)
```

Causes Igor to do an immediate update.

An update consists of:

- Redrawing any windows that have been uncovered.
- Re-evaluating any dependency formulas (e.g., wave0 := K0 when K0 changes).
- Redrawing windows displaying objects (e.g., waves) that have changed.

Igor does updates automatically in its outer loop. You should call DoUpdate *only* if you want Igor to do an update *before* your XOP returns to Igor.

**NOTE**: If your XOP adds a window to Igor or if the update of an Igor window indirectly calls your XOP again, this callback will cause recursion. This can cause your XOP to crash if you do not handle it properly. See **Handling Recursion** on page 137 for details.

```
void
PauseUpdate(savePtr)
long* savePtr;              // Place to save current state of PauseUpdate
```

Tells Igor not to update graphs and tables until your XOP calls ResumeUpdate.

Make sure to balance this with a ResumeUpdate call.

```
void
ResumeUpdate(savePtr)
long* savePtr;              // Previous state from PauseUpdate call
```

Undoes effect of previous PauseUpdate.

Make sure this is balanced with a previous PauseUpdate call.

```
void
XOPNotice(noticePtr)
char* noticePtr;            // Message for Igor's history area
```

Displays the C string pointed to by noticePtr in Igor's history.

This is used mostly for debugging or to display the results of an operation.

**NOTE**: When Igor passes a message to you, you *must* get the message, using GetXOPMessage, and get all of the arguments, using GetXOPItem, *before* doing any callbacks, including XOPNotice. The reason for this is that the act of doing the callback overwrites the message and arguments that Igor is passing to you.

```
void
XOPResNotice(strListID, index)
int strListID;            // Resource ID to get string from
int index;                // String number in that resource
```

Gets a string from an STR# resource in the XOP's resource fork and displays it in Igor's history.

The resource must be of type 'STR#'. The resource ID should be between 1100 and 1199.

These resource IDs are reserved for XOPs.

strListID is the resource ID of the STR# containing the string.

index is the number of the string in the STR# resource.

```
int
WaveList(listHandle, match, sep, options)
Handle listHandle;        // Handle to contain list of waves
char* match;              // "*" for all waves or match pattern
char* sep;                // Separator character, normally ";"
char* options;            // Options for further selection of wave
```

Puts a list of waves from the current data folder that match the parameters into listHandle.

The function result is 0 if OK or an error code if the parameters were not legal or another problem (such as out of memory) occurred.

On input, listHandle is a handle to 0 bytes which you have allocated, typically with NewHandle.

Note that the text in the handle is *not* null terminated. Use GetHandleSize to find the number of bytes in the handle. To use C string functions on this text you need to copy it to a local buffer and null-terminate it or add a null terminator to the handle and lock the handle. See **Understand the Difference Between a String in a Handle and a C String** on page 321. If you pass the handle back to Igor, you must remove the null terminator and unlock the handle.

The meaning of the match, sep and options parameters is the same as for the built-in Igor WaveList function.

The handle must be allocated and disposed by the calling XOP.

```
int
WinList(listHandle, match, sep, options)
Handle listHandle;          // Handle to contain list of windows
char* match;                // "*" for all windows or match pattern
char* sep;                  // Separator character, normally ";"
char* options;              // Options for further selection of windows
```

Puts a list of windows that match the parameters into listHandle.

The function result is 0 if OK or an error code if the parameters were not legal or another problem (such as out of memory) occurred.

The meaning of the match parameter is the same as for the built-in Igor WinList function.

If options is "" then all windows are selected.

If options is "WIN:" then just the target window is selected.

If options is "WIN:typeMask" then windows of the specified types are selected.

The window type masks are defined in IgorXOP.h.

On input, listHandle is a handle to 0 bytes which you have allocated, typically with NewHandle.

Note that the text in the handle is *not* null terminated. Use GetHandleSize to find the number of bytes in the handle. To use C string functions on this text you need to copy it to a local buffer and null-terminate it or add a null terminator to the handle and lock the handle. See **Understand the Difference Between a String in a Handle and a C String** on page 321. If you pass the handle back to Igor, you must remove the null terminator and unlock the handle.

The handle must be allocated and disposed by the calling XOP.

```
int
PathList(listHandle, match, sep, options)
Handle listHandle;          // Handle to contain list of paths
char* match;                // "*" for all paths or match pattern
char* sep;                  // Separator character, normally ";"
char* options;              // Must be ""
```

Puts a list of symbolic paths that match the parameters into listHandle.

The function result is 0 if OK or an error code if the parameters were not legal or another problem (such as out of memory) occurred.

The meaning of the match parameter is the same as for the built-in Igor PathList function.

options must be "".

On input, listHandle is a handle to 0 bytes which you have allocated, typically with NewHandle.

Note that the text in the handle is *not* null terminated. Use GetHandleSize to find the number of bytes in the handle. To use C string functions on this text you need to copy it to a local buffer and

null-terminate it or add a null terminator to the handle and lock the handle. See **Understand the Difference Between a String in a Handle and a C String** on page 321. If you pass the handle back to Igor, you must remove the null terminator and unlock the handle.

The handle must be allocated and disposed by the calling XOP.

```
int
GetPathInfo2(pathName, fullDirPath)
const char* pathName;              // Input
char* fullDirPath[MAX_PATH_LEN+1];  // Output
```

pathName is the name of an Igor symbolic path.

Returns via fullDirPath the full native path to the directory referenced by pathName. The returned path includes a trailing colon on Macintosh and a trailing backslash on Windows.

The function result is 0 if OK or an error code if the pathName is not the name of an existing Igor symbolic path.

```
int
VariableList(listHandle, match, sep, varTypeCode)
Handle listHandle;         // Receives list of variable names
char* match;               // "*" for all variables or match pattern
char* sep;                 // Separator character, normally ";"
char* varTypeCode;         // Select which variable types to list
```

Puts a list of Igor global numeric variable names from the current data folder that match the parameters into listHandle.

The function result is 0 if OK or an error code if the parameters were not legal or another problem (such as out of memory) occurred.

match and sep are as for the WaveList callback.

varTypeCode is some combination of NT_FP32, NT_FP64 and NT_CMPLX. Use (NT_FP32 | NT_FP64 | NT_CMPLX) to get all variables. As of Igor Pro 3.0, all numeric global variables are double precision.

On input, listHandle is a handle to 0 bytes which you have allocated, typically with NewHandle. VariableList fills the handle  with text.s

Note that the text in the handle is *not* null terminated. Use GetHandleSize to find the number of bytes in the handle. To use C string functions on this text you need to copy it to a local buffer and null-terminate it or add a null terminator to the handle and lock the handle. See **Understand the Difference Between a String in a Handle and a C String** on page 321. If you pass the handle back to Igor, you must remove the null terminator and unlock the handle.

The handle must be allocated and disposed by the calling XOP.

```
int
StringList(listHandle, match, sep)
Handle listHandle;          // Receives list of string variable names
char* match;                // "*" for all strings or match pattern
char* sep;                  // Separator character, normally ";"
```

Puts a list of Igor global string variable names from the current data folder that match the parameters into listHandle.

The function result is 0 if OK or an error code if the parameters were not legal or another problem (such as out of memory) occurred.

match and sep are as for the WaveList callback.

On input, listHandle is a handle to 0 bytes which you have allocated, typically with NewHandle. StringList fills the handle with text.

Note that the text in the handle is *not* null terminated. Use GetHandleSize to find the number of bytes in the handle. To use C string functions on this text you need to copy it to a local buffer and null-terminate it or add a null terminator to the handle and lock the handle. See **Understand the Difference Between a String in a Handle and a C String** on page 321. If you pass the handle back to Igor, you must remove the null terminator and unlock the handle.

The handle must be allocated and disposed by the calling XOP.

```
int
CheckAbort(timeoutTicks)
long timeoutTicks;          // Ticks at which timeout occurs
```

Use this to check if it’s time to stop an operation or if user is pressing cmd-dot (Macintosh) or Ctrl-Break (Windows) to abort it.

Returns -1 if user is now pressing cmd-dot.

Returns 1 if timeoutTicks is not zero and TickCount > timeoutTicks. TickCount is the Macintosh tick counter. It increments approximately 60 times per second. The TickCount function is emulated by Igor when running on Windows.

Returns 0 otherwise.

CheckAbort this does a check only every .1 second no matter how often you call it.

```
int
DoCHIO(CHIOPtr)
CHIORecPtr CHIOPtr;         // Pointer to struct describing character I/O
```

This is used to do I/O with character-oriented devices.

You supply the address of your routine for reading characters, writing characters, checking how many characters are available for reading and returning unread characters to your input buffer. DoCHIO then does all the parameter parsing, reading or writing, conversion to/from ASCII, storing in variables or waves and all of the other details necessary to implement character-oriented read/write operations.

The CHIORec structure is documented in the IgorXOP.h file.

**NOTE**: New XOPs should not use DoCHIO. It was used in a complicated scheme that the old VDT and NIGPIB XOPs used to parse input. The idea was that the parsing would be done through a callback to Igor so that it would not need to be replicated in each XOP. However, this technique is no longer recommended because it makes program flow incomprehensible. Instead the new VDT2 and NIGPIB2 XOPs do their own parsing.

For examples using DoCHIIO, see the VDTRead, VDTReadWave, VDTWrite, and VDTWriteWave routines in the sample file VDTOperations.c in XOP Toolkit 3.1.

```
void
IgorError(title, errCode)
char* title;                // Short title for error alert
int errCode;                // Error code
```

Displays an error alert appropriate for the specified error code.

Title is a short string that identifies what generated the error.

errCode may be an Igor error code (defined in IgorXOP.h), an XOP-defined error code, or, when running on Macintosh, a Mac OS error code. To display a message for a Windows OS error, convert the code to an Igor code by calling WindowsErrorToIgorError.

Use this routine when an error occurs in an XOP but not during the execution of a command line operation, function or menu item routine. See **XOP Errors** on page 127 for details.

```
int
GetIgorErrorMessage(errCode, errorMessage)
int errCode;                // Error code
char errorMessage[256];     // Output string goes here
```

Returns via errorMessage the message corresponding to the specified error code.

errCode may be an Igor error code (defined in IgorXOP.h), an XOP-defined error code, or, when running on Macintosh, a Mac OS error code. To obtain a message for a Windows OS error,

convert the code to an Igor code by calling WindowsErrorToIgorError before caling GetIgorErrorMessage.

Do not pass 0 for errCode. There is no error message corresponding to 0.

errorMessage must be large enough to hold 255 bytes plus a trailing null.

The function result is 0 if OK, or a non-zero error code if the errCode parameter is invalid. If GetIgorErrorMessage fails to get a message, it sets *errorMessage to 0.

This routine is of use when you want to display an error message in your own window rather than in a dialog. See **XOP Errors** on page 127 for details.

```
int
SpinProcess(void)
```

Spins the beachball cursor and, on Macintosh, gives other programs a chance to do background processing.

Igor may be moved from the foreground to the background or vice versa when SpinProcess is called.

Returns non-zero if the user pressed cmd-dot (Macintosh) or Ctrl-Break (Windows) recently or zero otherwise.

To spin the beachball cursor without allowing background processing on Macintosh, call SpinCursor.

**NOTE**: If your XOP adds a window to Igor or if the update of an Igor window indirectly calls your XOP again, this callback will cause recursion. This can cause your XOP to crash if you do not handle it properly. See **Handling Recursion** on page 137 for details.

```
void
PutCmdLine(cmd, mode)
char* cmd;
int mode;
```

This is a lower level call than the FinishDialogCmd call which should be used in most cases.

Puts the text in the C string cmd into Igor's command line using the specified mode.

Liberal names of waves and data folders must be quoted before using them in the Igor command line. Use PossiblyQuoteName when preparing the command to be executed so that your XOP works with liberal names.

Modes are:

| | |
|---|---|
| INSERTCMD | Inserts text at current insertion point. |
| FIRSTCMD | Inserts text in front of command buffer. |
| FIRSTCMDCRHIT | Inserts text in front of command buffer, set crHit. |
| REPLACEFIRSTCMD | Replaces first line of command buffer with text. |
| REPLACEALLCMDSCRHIT | Replaces all lines and set crHit. |
| REPLACEALLCMDS | Replaces all lines of command buffer with text. |

The intended use for PutCmdLine is to put a command generated by an XOP dialog into Igor's command line. For this, use the FIRSTCMD mode to just put the command in the command line or the FIRSTCMDCRHIT mode to put the command in the command line and start execution of the command buffer. crHit is a variable within Igor itself which enables execution of commands in the command buffer. Execution of commands occurs in Igor's idle loop.

If you just want to submit a command to Igor for immediate execution, use XOPCommand or XOPSilentCommand instead.

```
int
IgorVersion(void)
```

Returns 100 times the version of Igor that is running.

For example, if the version of Igor is 5.01, IgorVersion returns 501.

You can check the Igor version during initialization of your XOP. If the version is not recent enough to run your XOP, return an error (using SetXOPResult) indicating that your XOP needs a later version of Igor.

**NOTE**: You should use the igorVersion global rather than IgorVersion. Calling IgorVersion will do no harm but it is slow.

```
int
XOPDisplayHelpTopic(title, topicStr, flags)
const char* title;          // Title for help window
const char* topicStr;       // The help topic to be displayed
long flags;
```

Displays help from an Igor help file for the specified topic. See **Igor Pro Help File** on page 292 for background information.

The title parameter is used only for modal help and supplies the title for a modal dialog containing the help. Modal help is described below.

topicStr is a help topic string that matches a help topic or subtopic in an Igor help file. Igor first searches open help files for the topic. If it is not found, Igor then searches all Igor help files in the folder containing the XOP file and subfolders. If it is still not found Igor then searches all Igor help files in the Igor Pro folder and subfolders.

The help file must be compiled in order for Igor to find the topic. Each time you open a file as a help file, Igor checks to see if it is compiled and if not asks if you want to compile it.

topicStr may have one of the following formats:

| Format | Example |
|---|---|
| <topic name> | "GBLoadWave XOP" |
| <subtopic name> | "The Load General Binary Dialog" |
| <topic name>[<subtopic name>] | "GBLoadWave XOP[The Load General Binary Dialog]" |

If the topic that you want to display is a subtopic, you should use the last form since it minimizes the chance that Igor will find another help file with the same subtopic name. Also, you must

choose descriptive topic and subtopic names to minimize the chance of a conflict between two help files.

Note that once you reference a help topic or subtopic from your executable code, you must be careful to avoid changing the name of the topic or subtopic.

The flags parameter is interpreted bitwise as follows:

| | |
|---|---|
| Bit 0 | If cleared, Igor displays non-modal help. If set, Igor displays modal help. |
| Bit 1 | If cleared, during modal help Igor displays the entire help file (if it is not too big) in the modal help dialog, with the specified topic initially in view. If set, during modal help Igor displays just the specified topic. |
| Bit 2 | If cleared, if the topic can not be found Igor displays an error dialog. If set, if the topic can not be found Igor does not display an error dialog. |
| All other bits | Reserved. You must pass zero for these bits. |

You *must* set bit 0 if you call XOPDisplayHelpTopic from a modal dialog. This causes Igor do display a dialog containing help on top of your dialog. If you fail to set bit zero when calling XOPDisplayHelpTopic from a modal dialog, Igor may behave erratically. Unfortunately, links in help files don't work during modal help.

If you are calling XOPDisplayHelpTopic in a non-modal situation, it is appropriate to clear bit zero, but not required. If you clear bit zero, Igor displays a normal Igor help file. If you set bit zero, Igor displays a modal help dialog.

You must set all other bits to zero.

Function result is 0 if OK or a non-zero code if the topic can not be found or some other error occurs.

```
int
XOPSetContextualHelpMessage(theWindow, msg, r)
XOP_WINDOW_REF theWindow;        // WindowRef on Mac, HWND on Windows
const char* msg;                 // The tip.
const Rect* r;                   // Hot rectangle.
```

Displays a message in the Igor Tips help window on Macintosh or in the status bar on Windows. Call this when your window is active and the user moves the cursor over an icon or other area of the window about which you have something to say.

theWindow is your WindowRef on Macintosh or your HWND on Windows. This refers to the window containing the control or icon for which you are providing help.

message is a C string containing the message to display.

r is a pointer to a Macintosh rectangle, even on Windows, that indicates the area of the window that the icon occupies. When the user moves the cursor out of this rectangle, Igor will remove the message. On Macintosh, this rectangle is in the local coordinates of the window containing the control or icon. On Windows, it is in client coordinates of the window containing the control or icon. On Windows, use WinRectToMacRect to translate the Windows RECT into a Macintosh Rect.

Function result is 0 if OK or IGOR_OBSOLETE.

The WindowXOP1 sample XOP illustrates the use of this function.

Added for Igor Pro 4.05 Carbon. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

```
int
IsMacOSX(void)
```

Returns the truth that we are running on Mac OS X.

This routine is available on Macintosh only.

```
int
WinInfo(index, typeMask, name, winPtr)
int index;                 // Index number of window
int typeMask;              // Code for type of window of interest
char name[MAX_OBJ_NAME+1]; // C string to receive name
XOP_WINDOW_REF* windowRefPtr;    // Pointer to WindowPtr or HWND.
```

Returns information about an Igor target window (graph, table, layout, notebook or control panel).

index and typeMask are inputs.

name and windowRefPtr and the function result are outputs.

On Macintosh, windowRefPtr points to a WindowPtr. On Windows, it points to an HWND.

index is an index starting from 0 for the top window, 1 for the next window and so on.

typeMask is a combination of GRAF_MASK , SS_MASK, PL_MASK, MW_MASK and PANEL_MASK for graphs, tables, page layouts, notebooks and control panels. Window types and type masks are defined in IgorXOP.h.

WinInfo stores the name of the specified window in name and stores the XOP_WINDOW_REF for the specified window in *windowRefPtr.

WinInfo returns the Igor window type of the specified window or 0 if no such window exists. If 0 is returned then the name and *windowRefPtr are undefined.

```
int
SaveXOPPrefsHandle(prefsHandle)
Handle prefsHandle;        // Handle containing prefs data.
```

Saves the handle in Igor's preferences file. You can retrieve the handle using GetXOPPrefs-Handle.

Igor makes a copy of the data in the handle, so the handle is still yours after you call this. Keep or dispose of it as you wish.

If you pass NULL for the prefsHandle parameter, Igor removes any existing XOP preferences from the Igor  preferences file.

Igor uses the name of your XOP's file to distinguish your preferences from the preferences of other XOPs.

Each time you call this routine, the Igor preferences file is opened and closed. Therefore, it is best to call each of it only once. One way to do this is to call GetXOPPrefsHandle when your XOPs starts and SaveXOPPrefsHandle when you receive the CLEANUP message.

The function result is 0 if OK or a non-zero error code.

As of this writing, XOP preference handles are stored in the Igor preferences file but this may change in the future.

```
int
GetXOPPrefsHandle(Handle* prefsHandlePtr)
Handle* prefsHandlePtr;
```

Retrieves your XOP's preference handle from the Igor preferences file, if you have previously stored it there using SaveXOPPrefsHandle. In this case, on return, *prefsHandlePtr will be your preferences handle and the function result will be 0. This handle is allocated by Igor but belongs to you to keep or dispose as you wish.

If the Igor preferences file does not contain your preferences, on return, *prefsHandlePtr will be NULL and the function result will be 0.

Igor uses the name of your XOP's file to distinguish your preferences from the preferences of other XOPs.

Each time you call this routine, the Igor preferences file is opened and closed. Therefore, it is best to call each of it only once. One way to do this is to call GetXOPPrefsHandle when your XOPs starts and SaveXOPPrefsHandle when you receive the CLEANUP message.

The function result is 0 if OK or a non-zero error code.

If the result is zero and *prefHandlePtr is not NULL then *prefsHandlePtr contains a handle to your preferences.

If the result is zero and *prefHandlePtr is NULL then there was no preferences data for your XOP. You should use default settings.

If the result is non-zero then an error occurred while trying to access preferences. *prefHandlePtr will be NULL and you should use default settings.

As of this writing, XOP preference handles are stored in the Igor preferences file but this may change in the future.

```
int
GetPrefsState(prefsStatePtr)
long* prefsStatePtr;        // Receives preferences state flag.
```

Returns via bit 0 of prefsStatePtr the truth that preferences are on. Other bits are reserved for future use.

See the Igor Pro manual for information about the preferences on/off state.

The function result is 0 if OK or an error code.

# Programming Utilities

```
int
MemClear(void *p, long numBytes)
void* p;                    // Pointer to memory to be cleared.
long numBytes;              // Number of bytes to be cleared.
```

Sets the specified number of bytes at the memory location pointed to by p to zero.

```
int
GetCStringFromHandle(h, str, maxChars)
Handle h;                   // Handle containing text
char* str;                  // Output C string goes here
int maxChars;               // Max number of characters before null
```

h is a handle containing a string.

str is a C string (null-terminated character array).

maxChars is the maximum number of bytes that str can hold, excluding the null terminator byte.

GetCStringFromHandle transfers the characters from h to str and null-terminates str.

If h is NULL, GetCStringFromHandle returns USING_NULL_STRVAR. This is typically a programmer error.

If the characters in h will not fit in str, GetCStringFromHandle returns STR_TOO_LONG.

If the characters fit, it returns 0.

For a discussion of C strings versus text handles, see **Understand the Difference Between a String in a Handle and a C String** on page 321.

```
int
PutCStringInHandle(str, h)
const char* str;            // Input C string
Handle h;                   // Handle to hold text
```

str is a C string (null-terminated character array).

h is a handle in which the C string data is to be stored.

PutCStringInHandle transfers the characters from str to h. Note that the trailing null from the C string is not stored in the handle.

If h is NULL, it returns USING_NULL_STRVAR. This is typically a programmer error.

If an out-of-memory occurs when resizing the handle, it returns NOMEM. If the operation succeeds, it returns 0.

For a discussion of C strings versus text handles, see **Understand the Difference Between a String in a Handle and a C String** on page 321.

```
int
CmpStr(char *str1, char *str2)
char* str1;                // C string
char* str2;                // C string
```

Does case-insensitive comparison.

Returns 0 if the strings are the same except for case. Returns -1 if str1 is alphabetically before str2 or 1 if str1 is alphabetically after str2.

```
int
strchr2(const char* str, int ch)
const char* str;           // C string to be searched
int ch;                    // Single-byte character to search for
```

strchr2 is like the standard C strchr function except that it is Asian-language-aware and assumes that the system default character encoding governs the path.

It returns a pointer to the first occurrence of ch in the null-terminated string str or NULL if there is no such occurrence. ch is a single-byte character.

On a system that uses an Asian script system as the default script, strchr2 knows about two-byte characters. For example, if you are searching for a backslash in a full path and if the path contains Asian characters, and if the second byte of an Asian character has the same code as the backslash character, strchr will mistakenly find this second byte while strchr2 will not.

On a system that does not use an Asian script system as the default script, strchr2 is just like strchr.

```
int
strrchr2(const char* str, int ch)
const char* str;           // C string to be searched
int ch;                    // Single-byte character to search for
```

strrchr2 is like the standard C strrchr function except that it is Asian-language-aware and assumes that the system default character encoding governs the path.

Returns a pointer to the last occurrence of ch in the null-terminated string str or NULL if there is no such occurrence. ch is a single-byte character.

On a system that uses an Asian script system as the default script, strrchr2 knows about two-byte characters. For example, if you are searching for a backslash in a full path and if the path contains Asian characters, and if the second byte of an Asian character has the same code as the backslash character, strrchr will mistakenly find this second byte while strrchr2 will not.

On a system that does not use an Asian script system as the default script, strrchr2 is just like strrchr.

```
int
IsINF32(floatPtr)
float* floatPtr;             // The number to test
```

Returns 1 if the number pointed to by floatPtr is +infinity or –infinity, 0 otherwise.

```
int
IsINF64(doublePtr)
double* doublePtr;           // The number to test
```

Returns 1 if the number pointed to by doublePtr is +infinity or –infinity, 0 otherwise.

```
int
IsNaN32(floatPtr)
float* floatPtr;             // The number to test
```

Returns 1 if the number pointed to by floatPtr is a NaN (not-a-number) or 0 otherwise.

```
int
IsNaN64(doublePtr)
double* doublePtr;           // The number to test
```

Returns 1 if the number pointed to by doublePtr is a NaN (not-a-number) or 0 otherwise.

```
int
SetNaN32(floatPtr)
float* floatPtr;
```

Sets the float pointed to by the parameter to NaN (not-a-number).

```
int
SetNaN64(doublePtr)
double* doublePtr;
```

Sets the double pointed to by the parameter to NaN (not-a-number).

```
int
DateToIgorDateInSeconds(numValues, year, month, dayOfMonth, secs)
int numValues;              // Number of dates to convert
short* year;                // e.g., 2004
short* month;               // 1=January, 2=February, . . .
short* dayOfMonth;
double* secs;               // Output in Igor date format
```

Converts dates into Igor date format (seconds since 1/1/1904).

numValues is the number of dates to convert.

year, month and dayOfMonth and secs are arrays allocated by the calling routine. The size of each array is specified by numValues.

On input, year, month and dayOfMonth hold the input values. On return, secs holds the output values.

The function result is zero or an error code.

This routine requires Igor Pro 5.00 or later. Earlier versions will return IGOR_OBSOLETE.

```
int
IgorDateInSecondsToDate(numValues, secs, dates)
int numValues;              // Number of dates to convert
double* secs;               // Input in Igor date format
short* dates;               // Output goes here
```

Converts dates in Igor date format (seconds since 1/1/1904) into date records.

numValues is the number of Igor dates to convert.

secs is an array of dates in Igor date format. Its size is specified by numValues.

dates is an array of shorts. It must hold 7*numValues shorts. For each input value, 7 values are written to the dates array, in the following order:

```
year, month, dayOfMonth, hour, minute, second, dayOfWeek
```

The function result is zero or an error code.

For example:

```
double secs[2];
short dates[2*7];
int err;

secs[0] = 0;                // Represents January 1, 1904, midnight.
secs[1] = 24*60*60;         // Represents January 2, 1904, midnight.
err = IgorDateInSecondsToDate(2, secs, dates);
```

This routine requires Igor Pro 5 or later. Earlier versions will return IGOR_OBSOLETE.

```
int
MoveLockHandle(theHandle)
void* theHandle;            // The handle to be moved and locked
```

Moves the handle to the top of the heap and locks it. Use this before dereferencing the handle to make sure the block of memory that the handle refers to is not relocated.

The function result is the state of the handle before it was locked. You can use this to later restore the handle to its previous state, using HSetState.

**NOTE**: In general, you should not lock handles if you can avoid it. Locking a handle fragments the heap or takes a lot of time (if you also move it high as you should) and makes it necessary for you to keep track of the state of the handle. However, if you are in doubt, it is better to lock the handle than to risk a crash. See **Dangling Pointer / Heap Scramble Problems** on page 312 for details.

```
void
XOPBeep(void)
```

Emits a beep.

```
void
XOPOKAlert(title, message)
const char* title;          // Dialog title.
const char* message;        // Message to be displayed.
```

Displays an alert (also known as "message box") and waits for the user to click OK.

```
int
XOPOKCancelAlert(title, message)
const char* title;          // Dialog title.
const char* message;        // Message to be displayed.
```

Displays an alert (also known as "message box") and waits for the user to click OK or Cancel.

Returns 1 for OK, -1 for cancel.

```
int
XOPYesNoAlert(title, message)
const char* title;          // Dialog title.
const char* message;        // Message to be displayed.
```

Displays an alert (also known as "message box") and waits for the user to click Yes or No.

Returns 1 for yes, 2 for no.

```
int
XOPYesNoCancelAlert(title, message)
const char* title;          // Dialog title.
const char* message;        // Message to be displayed.
```

Displays an alert (also known as "message box") and waits for the user to click Yes or No or Cancel.

Returns 1 for yes, 2 for no, -1 for cancel.

```
void
MacRectToWinRect(mr, wr)
const Rect* mr;             // Macintosh rectangle
RECT* wr;                   // Windows rectangle
```

Igor sometimes passes a Macintosh rectangle to your XOP as a message argument. Use MacRectToWinRect when you receive a Macintosh rectangle but you need a Windows rectangle.

```
void
WinRectToMacRect(wr, mr)
const RECT* wr;             // Windows rectangle
Rect* mr;                   // Macintosh rectangle
```

Some XOPSupport routines take a Macintosh rectangle as a parameter. Use WinRectToMacRect when an XOPSupport routine requires a Macintosh rectangle but you have a Windows rectangle.

# Macintosh Emulation Routines

These Mac OS API routines are emulated by Igor when running on Windows so that XOPs can share memory and menus with Igor and to make it easier to write a cross-platform XOP with a single set of source code. In many cases, the emulation is not complete but rather just sufficient for these purposes. This section describes the behavior of the routines as they are emulated by Igor on Windows.

## Emulated Macintosh Memory Management Routines

When dealing with memory objects that are shared between Igor and your XOP, you must use Macintosh memory management routines, described in this section, even when you are running on Windows. When dealing with your own private data, you can use Macintosh routines, standard C memory management routines, or Windows memory management routines. Using the Macintosh routines makes it easier to write cross-platform XOPs.

When running on Macintosh, these routines are implemented by the Mac OS. When running on Windows, they are supplied by the IGOR.lib, with which all XOPs are linked.

See **Data Sharing** on page 139 for an overview.

```
Ptr
NewPtr(size)
long size;
```

NewPtr allocates a block of size bytes in the heap and returns a pointer to that block.

In the event of an error, NewPtr returns NULL.

After calling NewPtr, you can call MemError to see if there was a error during allocation of the memory. However, it is customary to assume that the error is NOMEM if NewPtr returns NULL. For example:

```
Ptr p;
p = NewPtr(100);
if (p == NULL)
   return NOMEM;
```

You must call DisposePtr to free the block of memory allocated by NewPtr.

```
long
GetPtrSize(p)
Ptr p;
```

p is a pointer to a block of memory allocated by NewPtr.

GetPtrSize returns the size in bytes of that block of memory.

```
void
SetPtrSize(p, size)
Ptr p;
long size;
```

p is a pointer to a block of memory allocated by NewPtr.

SetPtrSize resizes the block to the number of bytes specified by the size parameter.

SetPtrSize can fail. You must call MemError after calling SetPtrSize to verify that the resizing succeded.

```
void
DisposePtr(p)
Ptr p;
```

p is a pointer to a block of memory allocated by NewPtr.

DisposePtr frees the block of memory. After calling DisposePtr, you must not access the memory.

```
Handle
NewHandle(size)
long size;
```

NewHandle allocates a block of size bytes in the heap and returns a reference to that block.

In the event of an error, NewHandle returns NULL.

After calling NewHandle, you can call MemError to see if there was a error during allocation of the memory. However, it is customary to assume that the error is NOMEM if NewHandle returns NULL. For example:

```
Handle h;
h = NewHandle(100);
if (h == NULL)
   return NOMEM;
```

You must call DisposeHandle to free the block of memory allocated by NewHandle

```
int
HandToHand(destHandlePtr)
Handle* destHandlePtr;
```

HandToHand creates a new block of memory that contains the same data as an existing block of memory.

It returns 0 if the allocation succeded or an error code.

In this example, we assume that h1 is an existing handle to a block of memory.

```
Handle h2;
int err;
h2 = h1;              // h2 now refers to the same block of memory as h1.
if (err = HandToHand(&h2))
   return err;
<Use h2>              // h2 now refers to a new block of memory.
DisposeHandle(h2);
```

```
int
HandAndHand(h1, h2)
Handle h1;
Handle h2;
```

h1 and h2 are handles returned by NewHandle.

HandAndHand appends the contents of the block of memory referenced by h1 to the block of memory referenced by h2. In the process, it resizes the block referenced by h2.

It returns 0 if the allocation succeded or an error code.

```
long
GetHandleSize(h)
Handle h;
```

h is a handle referencing a block of memory in the heap.

GetHandleSize returns the number of bytes in the block of memory that h references.

```
void
SetHandleSize(h, size)
Handle h;
long size
```

h is a handle referencing a block of memory in the heap.

SetHandleSize resizes the block of memory that h references to the number of bytes specified by size.

SetHandleSize can fail. You must call MemError after calling SetHandleSize to verify that the resizing succeded.

```
void
DisposeHandle(h)
Handle h;
```

h is a handle referencing a block of memory in the heap.

DisposeHandle frees the block of memory that h references. After calling DisposeHandle, h is no longer a valid reference.

```
int
HGetState(h)
Handle h;
```

h is a handle referencing a block of memory in the heap.

HGetState returns the state of the handle. HGetState is used in conjunction with HSetState to save and restore the locked/unlocked state of a handle. Usually MoveLockHandle is used instead of HGetState.

```
void
HSetState(h, state)
Handle h;
int state;
```

h is a handle referencing a block of memory in the heap.

HSetState sets the state of the handle. HSetState is used in conjunction with HGetState to save and restore the locked/unlocked state of a handle.

```
void
HLock(h)
Handle h;
```

h is a handle referencing a block of memory in the heap.

HLock changes the state of the handle to locked, so that the block of memory to which the handle refers will not be relocated by the memory manager.

Rather than unconditionally locking and unlocking a block of memory, it is usually better to call MoveLockHandle to lock the handle and HSetState to restore it.

```
void
HUnlock(h)
Handle h;
```

h is a handle referencing a block of memory in the heap.

HUnlock changes the state of the handle to unlocked, so that the block of memory to which the handle refers can be relocated by the memory manager.

Rather than unconditionally locking and unlocking a block of memory, it is usually better to call MoveLockHandle to lock the handle and HSetState to restore it.

```
void
MoveHHi(h)
Handle h;
```

h is a handle referencing a block of memory in the heap.

MoveHHi moves the block of memory referenced by h to the top of the heap. It is used prior to HLock to avoid heap fragmentation.

The MoveLockHandle routine calls MoveHHi and then HLock and is usually used instead of calling them separately.

```
int
PtrToHand(srcPtr, destHandlePtr, size)
Ptr srcPtr;
Handle* destHandlePtr;
long size;
```

PtrToHand allocates a new block of memory in the heap and then copies size bytes from srcPtr to the new block. It returns a handle referencing the new block via destHandlePtr.

It returns 0 if the allocation succeded or an error code.

```
int
PtrAndHand(p, h, size)
Ptr* p;
Handle h;
long size;
```

h is a handle referencing a block of memory in the heap.

PtrAndHand resizes the block and then appends size bytes of data to the block by copying from the pointer p.

It returns 0 if the allocation succeded or an error code.

```
int
MemError(void)
```

MemError returns the error status from the most recently called memory management routine.

Use MemError after calling a memory management routine, such as SetPtrSize, that does not directly return an error status.

## Emulated Menu Management Routines

XOP menu items appear in Igor menus and XOPs can add menus to the Igor menu bar. On Windows, Igor uses Macintosh emulation to implement menus. Therefore, XOPs must use Macintosh menu manager routines to manipulate menu items and menus.

See **Menu Manager Routines** on page 232 for an overview.

```
short
CountMItems(theMenu)
MenuHandle theMenu;
```

Returns the number of menu items in the specified menu.

```
void
DeleteMenuItem(theMenu, itemNumber)
MenuHandle theMenu;
short itemNumber;
```

Deletes the specified menu item from the menu.

itemNumber is one-based. The first item in the menu is number 1.

You can delete items from your own menus but you must never delete items from Igor's menus.

```
void
insertmenuitem(menuH, itemString, afterItem)
MenuHandle menuH;
char* itemString;
short afterItem;
```

Inserts a new item into the menu after the item specified by afterItem.

itemString is the text for the new item.

afterItem is one-based. If you pass 1, the new item will be inserted after the first item in the menu. If afterItem is greater than the number of any existing item, the new item is appended to the end of the menu. If afterItem is 0, the new item is inserted at the beginning of the menu.

You can insert items into your own menus but you must never insert items into Igor's menus.

```
void
appendmenu(menuH, itemString)
MenuHandle menuH;
char* itemString;
```

Adds a new item to the end of the menu.

itemString is the text for the new item.

You can add items to your own menus but you must never add items to Igor's menus.

```
void
getmenuitemtext(theMenu, itemNumber, itemString)
MenuHandle theMenu;
short itemNumber;
char* itemString;
```

Retrieves the text for the specified item.

itemNumber is one-based. The first item in the menu is number 1.

The item text is returned as a C string via itemString.

```
void
setmenuitemtext(menuH, itemNumber, itemString)
MenuHandle menuH;
short itemNumber;
char* itemString;
```

Sets the text for the specified item.

itemNumber is one-based. The first item in the menu is number 1.

itemString is the new text for the item.

You can set the text for your own items but you must never set the text for Igor's items.

```
void
DisableItem(menuH, itemNumber)
MenuHandle menuH;
short itemNumber;
```

Disables (grays out) the specified menu item.

itemNumber is one-based. The first item in the menu is number 1.

You can disable items from your own menus but you must never disable items from Igor's menus.

```
void
EnableItem(menuH, itemNumber)
MenuHandle menuH;
short itemNumber;
```

Enables the specified menu item.

itemNumber is one-based. The first item in the menu is number 1.

You can enable items from your own menus but you must never enable items from Igor's menus.

```
void
CheckItem(theMenu, itemNumber, checked)
MenuHandle theMenu;
short itemNumber;
int checked;
```

Adds a check mark to or removes a check mark from the specified menu item.

itemNumber is one-based. The first item in the menu is number 1.

You can call this routine on your own menu items but you must never call it on Igor's menu items.

## Miscellaneous Emulated Macintosh Routines

```
unsigned long
TickCount(void)
```

On Macintosh, TickCount returns a count of the number of ticks that have elapsed since the Mac OS started up. On Windows, it returns a count of the number of ticks that have elapsed since Igor started up.

A tick is approximately one sixtieth of a second. TickCount is a simple way to determine the amount of time that has elapsed from one point in the program to another.

*Appendix* A

# XOP Toolkit 5 Upgrade Notes

Overview .......... 507
Changes You Must Make .......... 507
Change Your XOPI Resource .......... 508
XOP Toolkit 5 New Features .......... 509
Reorganization Of XOP Toolkit Folders .......... 510
Reorganizing Your XOP Folders .......... 512
    Reorganizing a CodeWarrior Pro 8 Project Folder .......... 512
    Reorganizing a Visual C++ 6 Project Folder .......... 513
    Reorganizing a Visual C++ 7 Project Folder .......... 514
Structure Alignment .......... 515
XOPSupport Additions .......... 516
Apple-Related Changes .......... 518
    Apple Resource Definition Syntax .......... 518
    LITTLE_ENDIAN Symbol .......... 518
    DOUBLE and double .......... 518

# Overview

These notes are for XOP programmers transitioning from XOP Toolkit 3.1 to XOP Toolkit 5.

XOP Toolkit Version 5 provides support and samples for creating XOPs using these development systems:

For Mac OS XOPs:
Metrowerks' CodeWarrior Pro 8.3 or later (Mac OS 9 or Mac OS X)
Apple's Xcode (Mac OS X 10.3 or later)

For Windows XOPs:
Microsoft's Visual C++ 6
Microsoft's Visual C++ 7 (known as Visual C++.NET)

The CodeWarrior Pro 8.3 project also work with "CodeWarrior Development Studio v9 for Mac OS" which is Metrowerks' latest Macintosh development system.

If you are an experienced programmer then you should be able to compile XOPs using other development systems after reading the XOP Toolkit documentation on setting up projects for the systems listed above.

XOP Toolkit 5 can create XOPs compatible with Igor Pro 4 or Igor Pro 5. Your XOP will run under Igor Pro 4 only if you avoid Igor Pro 5-specific features, which are listed below.

The support and sample files for XOP Toolkit 5 come in a folder named IgorXOPs5. If you have existing XOP projects, it is recommended that you *copy* them into the IgorXOPs5 folder. Leave your old projects as they are and do all new development in the new folder.

All of the new XOP Toolkit 5 features require Igor Pro 5. If your XOP must support Igor Pro 4, you must not use these new features. Since life is complicated enough already, we recommend that you do new XOP development for Igor Pro 5 and let any Igor Pro 4 users continue to use the old version of your XOP.

# Changes You Must Make

There are a few things that you must do to make your old XOPs work with XOP Toolkit 5:

1. Change your XOPI resource. This is described below under **Change Your XOPI Resource**.
2. Change your project to reference the new XOPSupport library. Depending on your development system, this will be one of the following:

| | |
|---|---|
| CodeWarrior | IgorXOPs5:XOPSupport:CW8:XOPSupport CFM.lib |
| Xcode | IgorXOPs5/XOPSupport/Xcode/XOPSupport.lib |
| Visual C++ 6 | IgorXOPs5\XOPSupport\VC6\XOPSupport.lib |
| Visual C++ 7 (.NET) | IgorXOPs5\XOPSupport\VC7\XOPSupport.lib |

3. Change the pragmas that control structure alignment. See **Structure Alignment** below.
4. If you have existing CodeWarrior CFM XOPs, double-check that the entry points specified in the PPC Linker settings pane of your project settings are correct. They should be as follows:

   ```
   Initialization: __initialize
   Main: main
   Termination: __terminate
   ```

   In previous versions of the XOP Toolkit, the initialization and termination entry points were left blank. This does not appear to cause a problem in C XOPs but it does cause a problem in C++ XOPs. At any rate, it is best to set the entry points for all CodeWarrior CFM XOPs.

## Change Your XOPI Resource

Igor looks at your XOP's 'XOPI' resource to get information about your XOP during Igor's initialization. For XOP Toolkit 5, change your Macintosh 'XOPI' resource (in, for example, XOP1.r) to this:

```
resource 'XOPI' (1100) {
   XOP_VERSION,              // XOP protocol version.
   DEV_SYS_CODE,             // Development system information.
   0,                        // Obsolete - set to zero.
   0,                        // Obsolete - set to zero.
   XOP_TOOLKIT_VERSION,      // XOP Toolkit version.
};
```

Change your Windows 'XOPI' resource (in, for example, XOP1WinCustom.rc) to this:

```
1100 XOPI
BEGIN
   XOP_VERSION,              // Version number of host XOP system.
   DEV_SYS_CODE,             // Development system information.
   0,                        // Obsolete - set to zero.
   0,                        // Obsolete - set to zero.
   XOP_TOOLKIT_VERSION       // XOP Toolkit version.
END
```

# XOP Toolkit 5 New Features

The process for creating an external command line operation has been simplified through the use of a new component of Igor Pro 5 called **Operation Handler**. Operation Handler does all of the parsing of command parameters for you. All you need do is to define your operation's syntax, register it when your XOP is launched, and supply the address of the function that Igor should call when your operation is invoked. See Operation Handler on page 151 for details.

External operations created using Operation Handler can be called from Igor user-defined functions. XOPs have to be rewritten to take advantage of this and in some cases (e.g., VDT and NIGPIB) the new XOPs might need different syntax from the old ones.

XOP Toolkit 5 and Igor Pro 5 support Apple's Mach-O binary executable format for XOPs. This makes it possible to use OS X frameworks from XOP projects. See Macintosh CFM Versus Mach-O on page 6 and Mach-O XOP Projects in CodeWarrior Pro 8 on page 75 or XOPs in Xcode on page 79 for details.

You can create XOPs using Apple's Xcode development system. Such XOPs will run with Igor Pro 5 or later. See XOPs in Xcode on page 79 for details.

XFUNCs can have pass-by-reference parameters. See Pass-By-Reference Parameters on page 198 for details.

XOPs can call user-defined functions and external functions added by other XOPs. See Calling User-Defined and External Functions on page 285 for details.

External operations and functions can receive Igor structures as parameters. This requires Igor Pro 5.03 or later. See Using Igor Structures as Parameters on page 281 for details.

XOP Toolkit 5 includes two new sample projects:

| | |
|---|---|
| NIGPIB2 | A revamp of NIGPIB which uses Operation Handler to supports calling external operations from Igor user functions. |
| VDT2 | A revamp of VDT which uses Operation Handler to supports calling external operations from Igor user functions. |

# Reorganization Of XOP Toolkit Folders

To make it easier to support multiple development systems, the organization of the XOP Toolkit folders has been changed. Previously an XOP folder looked something like this:

```
XOP1
   XOP1.c
   XOP1.r                  // Macintosh resource file
   XOP1 Mac                // CodeWarrior Project file
   XOP1 Mac Data           // CodeWarrior Project folder
   XOP1.xop                // Compiled XOP file
   XOP1 Help.ihf           // XOP help file
```

or like this:

```
XOP1
   XOP1.c
   XOP1.rc                 // Windows resource file
   XOP1.dsp                // Visual C++ 6 Project file
   XOP1.dsw                // Visual C++ 6 Workspace file
   XOP1.xop                // Compiled XOP file
   XOP1 Help.ihf           // XOP help file
```

With XOP Toolkit 5, the development-system-specific files have been moved into subdirectories. The resulting organization looks like this:

```
XOP1
   XOP1.c
   XOP1.r                  // Macintosh resource file
   XOP1.rc                 // Windows resource file
   XOP1 Help.ihf           // XOP help file

   CW8                     // Folder for CodeWarrior Pro 8 project files
      XOP1.mcp             // CodeWarrior Project file
      XOP1 Data            // CodeWarrior Project folder
      XOP1.xop             // Compiled XOP file

   Xcode                   // Folder containing Xcode project files
      XOP1.xcode           // Xcode project folder
      project.pbxproj      // Xcode project file
      XOP1.xop             // Compiled XOP file

   VC6                     // Folder for Visual C++ 6 project files
      XOP1.dsp             // VC6 Project file
      XOP1.dsw             // VC6 Workspace file
      XOP1.xop             // Compiled XOP file
```

```
VC7                     // Folder for Visual C++ 7 (.NET) project files
   XOP1.vcproj          // VC7 Project file
   XOP1.sln             // VC7 Solution file
   XOP1.xop             // Compiled XOP file
```

"VC7" refers to Microsoft's Visual C++ version 7, better known as Visual C++ .NET.

Note that the compiled XOP file is stored in the development system folder (e.g., VC6) so that you can compile the XOP on multiple development systems without conflict.

Note also that the XOP help file is not in the same folder as the executable XOP. Igor will not find the XOP help file there. You will have to move it into the same folder as the XOP itself.

The XOPSupport folder, to which all XOP projects refer, also now contains CW8, Xcode, VC6 and VC7 folders. The XOPSupport library files are in those folders.

# Reorganizing Your XOP Folders

If you are updating an existing XOP to use XOP Toolkit 5, you need to remove your old XOPSupport library and add the new XOPSupport library. The new XOPSupport library is in:

```
IgorXOPs5:XOPSupport:CW8
```

or

```
IgorXOPs5:XOPSupport:VC6
```

or

```
IgorXOPs5:XOPSupport:VC7
```

You do not need to do any further reorganization of your XOP folder. However, if you want to reorganize your XOP folder to be like the XOP Toolkit organization, which we recommend if there is any chance that you might use more than one development system, here is how you do it:

## Reorganizing a CodeWarrior Pro 8 Project Folder

1. Open your XOP folder.
2. Move the CodeWarrior project file and the associated data folder into a folder named CW8 inside your XOP folder as shown above for the XOP1 sample. Add the .mcp extension to the project file if necessary.
3. Open the project file in CodeWarrior Pro 8 or later.
4. Open the project settings window.
5. Change the access paths from:

   {Project}:

   {Project}::XOPSupport:

   to

   {Project}:: (The folder containing your .c and .h files.)

   {Project}:::XOPSupport: (The IgorXOPs5:XOPSupport folder.)
6. Save and close the project window.
7. Remove the old XOPSupport library and add the new one

   (IgorXOPs5/XOPSupport/CW8/XOPSupport CFM.lib).

## Reorganizing a Visual C++ 6 Project Folder

1. Open your XOP folder.
2. Move the VC6 project file (.dsp) and the associated workspace file (.dsw) into a folder named VC6 inside your XOP folder as shown above for the XOP1 sample.
3. Open the workspace file in Visual C++ 6.
4. Click the FileView tab in the Workspace window.
5. Remove the .rc file and add it back using Project->Add To Project->File.
6. Right-click the icon for each source (.c or .cpp) file, choose Properties, and change the "Persist As" path to the source file. For example, if the "Persist As" is:

   .\XOP1.c

   change it to:

   ..\XOP1.c
7. Change the "Persist As" path for the Igor.lib file to:

   ..\..\XOPSupport\Igor.lib
8. Remove the old XOPSupport library file (XOPSupportX86.lib) and add the new one (IgorXOPs5\XOPSupport\VC6\XOPSupport.lib).
9. Open the project settings window.
10. Choose All Configurations from the Settings For popup menu.
11. Click the C++ tab, choose the Preprocessor category and change

    ..\XOPSupport

    to

    ..\..\XOPSupport
12. Click the Resources tab and change

    ..\XOPSupport

    to

    ..\..\XOPSupport
13. Click OK to close the project settings dialog.

## Reorganizing a Visual C++ 7 Project Folder

1. Open your XOP folder.
2. Move the VC7 project file (.vcproj) and the associated solution file (.sln) into a folder named VC7 inside your XOP folder as shown above for the XOP1 sample.
3. Open the solution file in Visual C++ 7.
4. Click the FileView tab in the Workspace window.
5. Remove all of the source files and add them back using Project->Add To Project->File.
6. Remove the XOPSupport library file and add

    IgorXOPs5\XOPSupport\VC7\XOPSupport.lib.
7. Remove the Igor.lib library file and add

    IgorXOPs5\XOPSupport\Igor.lib
8. Open the project settings window.
9. Choose All Configurations from the Settings For popup menu.
10. Click the C++ tab, choose the Preprocessor category and change

    ..\XOPSupport

    to

    ..\..\XOPSupport
11. Click the Resources tab and change

    ..\XOPSupport

    to

    ..\..\XOPSupport
12. Click OK to close the project settings dialog.

# Structure Alignment

All structures passed between Igor and an XOP must be two-byte aligned. Otherwise a crash will occur. To ensure this alignment, you need to use pragmas to tell the compiler to align such structures on two byte boundaries.

In XOP Toolkit 3.1, this was done by putting something like this in your source files:

```
#if GENERATINGPOWERPC
   #pragma options align=mac68k
#endif
#ifdef _WINDOWS_
   #pragma pack(2)
#endif

. . . structure definitions here . . .

#if GENERATINGPOWERPC
   #pragma options align=reset
#endif
#ifdef _WINDOWS_
   #pragma pack()
#endif
```

With XOP Toolkit 5, use the following cleaner and functionally equivalent technique:

```
#include "XOPStructureAlignmentTwoByte.h"

. . . define structures here . . .

#include "XOPStructureAlignmentReset.h"
```

When you update your old XOPs to use XOP Tookit 5, replace the old #if statements with this new #include statement.

An early beta version of XOP Toolkit 5 used #defines named XOP_SET_STRUCT_PACKING and XOP_RESET_STRUCT_PACKING for this purpose. However, this turned out to be incompatible with the GCC compiler used in Apple's Xcode development system. If you are an early beta tester and you have these #defines in your code, use the new technique instead.

# XOPSupport Additions

These new XOPSupport routines are documented in Chapter 13. Unless otherwise noted, the new routines were added in XOP Toolkit release 5.00. Some of these routines require specific versions of Igor. See Chapter 13 for details.

| Routine | Description |
|---|---|
| GetCStringFromHandle | Converts from Igor string in handle to C string. |
| PutCStringInHandle | Converts from C string to Igor string in handle. |
| WaveLock | Supports Igor Pro 5 wave locking feature. |
| SetWaveLock | Supports Igor Pro 5 wave locking feature. |
| RegisterOperation | Used with Operation Handler. |
| SetOperationNumericVariable | Call only from Operation Handler “execute” function. |
| SetOperationStringVariable | Call only from Operation Handler “execute” function. |
| SetFileLoaderOperationOutput Variables | Use this in place of SetFileLoaderOutputVariables when using Operation Handler. It sets local variables when called from a user-defined function. |
| VarNameToDataType | Call only from Operation Handler “execute” function. |
| StoreNumericDataUsingVarName | Call only from Operation Handler “execute” function. |
| StoreStringDataUsingVarName | Call only from Operation Handler “execute” function. |
| GetFunctionInfo | Used to call a user-defined function from your XOP. |
| CheckFunctionForm | Used to call a user-defined function from your XOP. |
| CallFunction | Used to call a user-defined function from your XOP. |
| DateToIgorDateInSeconds | Returns a date in Igor format (seconds since 1/1/1904). |
| IgorDateInSecondsToDate | Converts a date from Igor format to day, month, year format |

| Routine | Description |
|---|---|
| GetCStringFromHandle | Converts from Igor string in handle to C string. |
| | format. |
| HFSToPosixPath | Converts HFS file path to Posix.<br>Added in release 5.03. |
| GetNVAR, SetNVAR, GetSVAR, SetSVAR | Allow external function to access structure fields.<br>Added in release 5.03. |
| GetLeafName | Returns leaf part of a file path.<br>Added in release 5.04. |
| GetTextWaveData, SetTextWaveData | Provide faster access to text wave data.<br>Added in release 5.04. |
| GetWaveDimensionLabels, SetWaveDimensionLabels | Provide faster access to wave dimension labels.<br>Added in release 5.04. |
| SetOperationWaveRef | Sets destination wave in external operation.<br>Added in release 5.04. |
| MDChangeWave2 | Redimensions wave without changing its data.<br>Added in release 5.04. |
| IsINF32, IsINF64 | Tests if number is infinity.<br>Added in release 5.04. |
| ScaleClipAndRound | Scales, clips and rounds array of data.<br>Added in release 5.04. |

# Apple-Related Changes

The following changes in Apple's header files may requires changes to your XOP.

## Apple Resource Definition Syntax

The file XOPStandardHeaders.r previously contained some #defines (e.g. #define DLOG_RezTemplateVersion 0) that allowed old syntax when defining resources of the following types: cfrg, ALRT, DLOG, wctb, WIND, PICT, clut. These #defines allowed XOP programmers to use old versions of Apple resource definitions.

To get up with the times, this version of XOPStandardHeaders.r omits those #defines. If you relied on them, you will get errors when compiling your resources. In most cases, the fix is to add a line to the resource definition. For example, here is a new resource definition with the added line indicated by a comment:

```
resource 'DLOG' (1260) {
   {50, 30, 430, 620},
   movableDBoxProc,
   invisible,
   noGoAway,
   0x0,
   1260,
   "Load General Binary",
   noAutoCenter            // <- Added this line.
};
```

The fix above applies to DLOG, ALRT and WIND resources. For details, consult Apple's documentation on resource definition formats or look at the definition of the resource in Apple's resource files (e.g., Dialogs.r., MacWindows.r, etc.).

## LITTLE_ENDIAN Symbol

Previously the file XOPStandardHeaders.h defined LITTLE_ENDIAN if and only if the compile was on a Windows system. Apple has now stolen that symbol from us so we have changed it to XOP_LITTLE_ENDIAN.

If your code relies on the LITTLE_ENDIAN defined in XOPStandardHeaders.h, you must change it to XOP_LITTLE_ENDIAN.

## DOUBLE and double

Previously the XOP Toolkit defined the symbol DOUBLE to mean double. This was left over from the days when a double on Macintosh could be 8 bytes or 10 bytes, depending on whether a program was compiled to use a math coprocessor. XOP Toolkit 5 no longer uses DOUBLE but it is still defined in IgorXOP.h so that old XOPs will continue to compile.

*Appendix* B

# Porting Macintosh XOPs to Carbon

Overview ........ 521
XOP Toolkit 3.13 Release Notes, June 28, 2002 ........ 522
Issues Relating To Carbon And Igor XOPs ........ 522
What You Need To Develop A Carbon XOP ........ 523
Porting An Existing Mac XOP To Carbon ........ 523
Changes To Existing XOPSupport Routines ........ 525
Obsolete XOP Messages ........ 527
Obsolete XOPSupport Globals ........ 527
Obsolete XOPSupport Routines ........ 527
New XOPSupport Routines ........ 530
Menu Items Added By STR# 1101 ........ 530
Dialogs Under Carbon ........ 531
- Moveable Dialog Box ........ 532
- Default and Cancel Dialog Items ........ 532
- Macintosh Dialogs With Popup Menus ........ 532
- Other Controls In Dialogs ........ 534

Carbon API Versus Classic Mac API ........ 535
Accessing Resources ........ 536
Other Issues ........ 537
Changes That Affect Windows XOPs ........ 537

# Overview

"Carbon" is the name of the Apple API that supports programming under Mac OS X as well as Mac OS 9. In order to run native on Mac OS X, WaveMetrics had to port Igor to the Carbon API, which is a subset of the original Macintosh API. We did this with Igor Pro 4.05. At that time there was a pre-Carbon Igor Pro 4 and a Carbon Igor Pro 4.

Igor Pro 5 and XOP Toolkit 5 are Carbon-only.

In order for a Macintosh XOP to run native on Mac OS X and to run with Igor Pro 5, it must also use the Carbon API. If you have a pre-Carbon XOP that you want to run with Igor Pro 5 on Macintosh, you must port it to Carbon. This appendix tells you what you need to do.

The following are the release notes provided with the XOP Toolkit 3.13 - the first version that supported Carbon XOPs. Keep in mind that these notes were written before Igor Pro 5 and XOP Toolkit 5 existed. "Igor" refers to Igor Pro 4. Some parenthetical comments have been added to clarify statements in the notes that are now out-of-date.

# XOP Toolkit 3.13 Release Notes, June 28, 2002

Read this section if you used used XOP Toolkit 3.12 to create Carbon XOPs.

In release 3.13, we have changed the construction of CodeWarrior Carbon XOPs. The sample XOPs and instructions in release 3.12 were based on sample CodeWarrior projects distributed with Apple's Carbon SDK. With this release, the Carbon XOPs are constructed to match CodeWarrior's stationery projects. Under the new setup, Apple's Carbon SDK and its Carbon Support folder are no longer needed.

If you have your own Carbon XOPs based on XOP Toolkit 3.12 samples, you might want to update your XOPs. First make a backup of your existing XOP files. Then follow these steps:

1. In the project settings window, display the Access Paths pane and remove the Carbon Support access path.
2. In the project settings window, display the C/C++ Language pane and change the prefix file to "MacHeadersCarbon.h".
3. If you are using C++, display the PPC Linker pane and set the Initialization field to __initialize. Set the Termination field to __terminate.
4. Save and close the project settings.
5. Add the following file to the ANSI Libraries section of the project window:

    MSL_All_Carbon.Lib

6. Remove the following files from the project window:

    MSL_Runtime_PPC.Lib
    MSL_C_Carbon.Lib
    MSL_C++_Carbon.Lib
    console.stubs

To remove the files, select them and then click and hold until a popup menu appears. Select Clear from the popup menu.

# Issues Relating To Carbon And Igor XOPs

Carbon is Apple's API which allows a developer to write a program that will run under Mac OS 9 and Mac OS X. Some original Mac OS API routines are not supported in Carbon and Carbon implements some new routines. In order to support Mac OS X, WaveMetrics has "carbonized" Igor Pro. "Carbonizing" means using Apple's Carbon header files, linking with Apple's CarbonLib instead of old libraries such as InterfaceLib, and restricting OS calls to the Carbon API.

Carbon Igor runs under Mac OS 9.1 and Mac OS X. It does not run under Mac OS 8.x. (Igor Pro 5 requires Mac OS 9.2 or later.)

Here are the main ramifications on XOPs of the conversion of Igor to a Carbon application.

1. The XOP Toolkit has been revised to use Carbon. New XOP development should use the Carbon XOP Toolkit, use the Carbon headers and link with CarbonLib. These XOPs will run with Igor Pro Carbon on Mac OS 9.1 and Mac OS X. They will not run with pre-Carbon Igor Pro.
2. Igor Pro Carbon does not supports 68K XOPs. If you rely on a 68K XOP, your choices are to stick with the pre-Carbon Igor or to port your XOP to PowerPC, using the Carbon XOP Toolkit.
3. For the most part, pre-Carbon PowerPC XOPs will continue to work with Igor Pro Carbon when running under Mac OS 9.1.
4. In order to run under Mac OS X, all XOPs must be carbonized.
5. Carbonized XOPs can run only with a carbonized version of Igor. Apple's Carbon library does not support running in a pre-Carbon application. It reports and error -2821.

## What You Need To Develop A Carbon XOP

To compile a Carbon XOP, you need the following:
XOP Toolkit Carbon
CodeWarrior Pro 7 or later
(XOP Toolkit 5 requires CodeWarrior Pro 8 or later, or Apple’s Xcode.)

## Porting An Existing Mac XOP To Carbon

Your old XOP project folder should be completely backed up or you should work on a completely new copy of it.

Open your old project in CodeWarrior Pro 7 or later.

If your old project was based on a WaveMetrics example, it may have three targets: 68K, PPC and FAT. You can see the targets by clicking the Targets tab of the CodeWarrior project window. Delete the 68K and FAT targets. They are not needed because Carbon does not support 68K code. (To delete an icon in the CodeWarrior project window, click and hold till you see a popup menu. Then choose Clear from the popup menu.)

Click the Files tab, open any folder icons, and delete any 68K libraries.

If there is a folder icon named FAT Target Files in the CodeWarrior project window, delete it.

In your CodeWarrior Settings window, under Access Paths, in the System Paths section, you should have just the following two access paths:

{Compiler}MacOS Support
{Compiler}MSL

In your CodeWarrior Settings window, under Runtime Settings, change the Host Application to point to your carbonized Igor Pro application file.

In your CodeWarrior Settings window, under PPC Target, change the File Name to <name of your XOP>.xop (e.g., XOP1.xop).

In your CodeWarrior Settings window, under C/C++ Language, set Prefix File to "MacHeadersCarbon.h" (without the quotes).

If this is a C++ XOP, in your CodeWarrior Settings window, under PPC Linker, set the Initialization field to __initialize. Set the Termination field to __terminate. (That change should be made for C XOPs as well as C++.)

Save and close the project settings window.

Old XOPs linked with a number of Apple libraries such as InterfaceLib and MathLib. You must replace all of these in your CodeWarrior project with CarbonLib and CarbonFrameworkLib, which you can find in:

"MacOS Support:Universal:Libraries:StubLibraries"

Add the CodeWarrior library MSL_All_Carbon.Lib and remove all other MSL libraries. At this point, a typical Carbon XOP will have the following libraries:

- Mac Libraries
  - CarbonLib
  - CarbonFrameworkLib
- ANSI Libraries
  - MSL_All_Carbon.Lib
- Igor Libraries
  - XOPSupport PPC.Lib

In your source code, any XOP code inside a XOP_GLOBALS_ARE_A4_BASED ifdef or an applec ifdef can be removed. This code ran on 68K only.

The LoadXOPSegs XOPSupport routine is obsolete and has been removed. If this appears in your source code (in your main function), remove it.

In your source code, replace GENERATINGPOWERPC (now obsolete) with TARGET_CPU_PPC.

In your source code, replace any call to FrontWindow with a call to GetActiveWindowRef. This is necessary because Igor Pro Carbon includes a floating help window. GetActiveWindowRef calls the Mac OS FrontNonFloatingWindow routine.

If your XOP creates a dialog based on WaveMetrics examples:

1. Search your source code for "XOP_WINDOW_REF theDialog" and replace it with "XOP_DIALOG_REF theDialog".
2. Search your source code for "SetDialogPort(savePort);" and replace it with "SetPort(savePort);".
3. If you have a calls to a routine named "PopMenu", delete them.
4. If you have a call to routines named "BoxItem", delete them.

See **Dialogs Under Carbon** below for further details.

When compiling with the Carbon Igor XOP, your XOP will have access to the Carbon API only. This API restriction is triggered by the fact that the XOPStandardHeaders.h file contains the following statement:

```
#define TARGET_API_MAC_CARBON 1
```

(In XOP Toolkit 5, this #define is done by system headers, not XOPSupport headers.)

If your XOP does just number-crunching, this API change will probably not affect you. If your XOP includes a user-interface, it will affect you. If you use unsupported APIs, you will get errors when you compile. Problems that are commonly encountered in XOPs are described below. If your XOP is advanced, you will need to read the document "Carbon Porting Guide", which is included with Apple's Carbon SDK, to finish carbonizing your XOP.

# Changes To Existing XOPSupport Routines

Changed DeleteMenuItems to WMDeleteMenuItems because Apple usurped DeleteMenuItems.

```
SelEditItem(theDialog, itemNumber)
```

Prior to XOP Toolkit 4.0 (Carbon), if itemNumber was 0, this routine used the currently-selected edit item. This feature did not fit in with the Carbon way of doing things and is no longer supported. The itemNumber parameter must be the item number of an edit text item.

```
SetDialogPort(DialogPtr theDialog)
```

Previously this was defined as returning a WindowPtr. Now it is defined as returning a CGrafPtr.

`SetDialogBalloonHelpID`

This function previously worked with the Apple Balloon Help Manager. Carbon does not support Balloon Help. However, this function now links with the contextual help window in Carbon Igor. As before, it does nothing on Windows.

`XOPOpenFileDialog`

This routine now uses the Macintosh Navigation Manager. The construction of the fileFilterStr parameter has changed and the fileIndexPtr parameter is now used on Macintosh as well as Windows.

Prior to Carbon, the fileFilterStr was a concatenation of zero or more Macintosh file types, like "TEXT" or "TEXTABCD". Now this parameter provides control over the Show popup menu which the Macintosh Navigation Manager displays in the Open File dialog. As a consequence, the fileFilterStr is constructed differently. For example, the string "Text Files:TEXT:.txt;All Files:****:;" results in two items in the Show popup menu.

The fileIndexPtr parameter provides a mechanism to save and restore the state of the Show popup menu between calls to XOPOpenFileDialog.

See the documentation for XOPOpenFileDialog for further details.

`XOPSaveFileDialog`

The fileFilterStr and fileIndexPtr parameters, which previously were not used on Macintosh, now specify the types of files that can be save, if you allow saving in more than one format. See the documentation for XOPSaveFileDialog for details.

`FileLoaderGetOperationFlags`

As of the Carbon XOP Toolkit, FileLoaderGetOperationFlags is no longer supported. It used working directory reference numbers, which are not supported by Carbon. Use FileLoaderGetOperationFlags2 instead.

`GetFullMacPathToDirectory`

Prior to Carbon, this routine accepted either a volume reference number and directory ID or a working directory reference number and zero. Carbon does not support working directory reference numbers, so this routine now supports only a volume reference number and directory ID.

`CreatePopMenu`

Prior to the Carbon XOP Toolkit, on Macintosh the item identified by the titleItemNumber parameter was highlighted when the user clicked on the popup menu. As of the Carbon XOP Toolkit, it is no longer used but must be present for backward compatibility.

# Obsolete XOP Messages

The LOAD and SAVE messages are obsolete and are no longer sent to XOPs. Use the LOADSETTINGS and SAVESETTINGS messages instead.

# Obsolete XOPSupport Globals

The following XOP Toolkit globals are obsolete and have been removed:
hasFPU    has68KHardwareFPU

# Obsolete XOPSupport Routines

The following XOP Toolkit routines are obsolete and have been removed.

`GetA4, SendXOPA4ToIgor`

These were used by 68K XOPs only and 68K XOPs are no longer supported.

`QDPointer`

Replace with accessor routines defined in QuickDraw.h such as GetQDGlobalsThePort.

`EditItem`

This routine did not fit in with the Carbon way of doing things. There is no equivalent Mac OS or XOPSupport routine.

`ReleaseMenu`

Apple created a routine also named ReleaseMenu. The old XOP Toolkit ReleaseMenu called the Apple ReleaseResource routine. However, this is no longer valid because Apple changed the behavior of GetMenu in Carbon. In the unlikely event that you used the XOP Toolkit ReleaseMenu routine, Apple's new ReleaseMenu is probably still appropriate.

`SetPopMenu`

Use CreatePopMenu instead.

`PopMenu`

This is not needed because the operating system pops popup menus up. If you use this routine, delete the call to it.

`GetSymbolicPath`

This used working directory reference numbers which are not supported by Carbon. Use GetSymb and GetName followed by GetPathInfo2 instead.

```
char fileDirPath[MAX_PATH_LEN+1];
int err;

if (GetSymb() != '=') {
   err = NOEQUALS;
}
else {
   err = GetName(pathName);
   if (err == 0)
      err = GetPathInfo2(pathName,fileDirPath);
}
```

`GetStandardFileVRefNumAndDirID, SetStandardFileVRefNumAndDirID`

The following Macintosh-only XOPSupport routines are no longer supported because there is no way to implement them under Carbon and they are of little use.

`GetStandardFilePath, SetStandardFilePath`

These Macintosh and Windows XOPSupport routines are no longer supported because there is no way to implement them under Carbon and they are of little use. The XOPOpenFileDialog and XOPSaveFileDialog routines provide similar functionality through their initialDir parameters.

`FileFullySpecified`

This used working directory reference numbers which are not supported by Carbon. Use GetFullPathFromSymbolicPathAndFilePath and/or FullPathPointsToFile instead.

`GetVRefNumAndDirIDFromFullPath`

This routine was provided only for the benefit of existing Macintosh XOPs. The implementation was not suitable for Carbon. New XOPs use platform-independent techniques and don't require this routine. If you used this routine and need it, see Apple's LocationFromFullPath routine in the file FullPath.c in the MoreFiles sample code supplied with the Carbon SDK.

`OpenFileReadOnly, StdGetFile, StdPutFile`

These routines used working directory reference numbers which are not supported by Carbon.

`XOPOpenResFile, XOPUseResFile, XOPCloseResFile`

These routines can not be supported on Mac OS X. See **Accessing Resources** below for further information.

`SetDItemProc`

This routine set the draw procedure for a UserItem in a dialog. Modern dialogs generally don't use UserItems because the Mac Appearance Manager provides sufficient standard controls. If you find that you do need this routine, call the Carbon SetDialogItem routine instead.

`BoxItem`

This routine turns a UserItem into a box. Modern dialogs generally don't use UserItems because the Mac Appearance Manager provides sufficient standard controls, such as group boxes. If you use this call, you need to change the DITL item into a GroupBox control (see Other Controls In Dialogs) and then remove the BoxItem call.

`XOPDialog`

The first parameter was previously of type ModalFilterProcPtr. It is now of type ModalFilterUPP.

XOPDialog now calls the Mac OS StdFilterProc to detect the user pressing return or escape. To support this, you must make the following calls after you call GetXOPDialog and before you call XOPDialog:

```
SetDialogDefaultItem(theDialog, <item number of your default button>);
SetDialogCancelItem(theDialog, <item number of your cancel button>);
SetDialogTracksCursor(theDialog, 1);
```

As of Carbon 1.3, calling SetDialogDefaultItem on a disabled button sets the button to the enabled state. This appears to be a Mac OS bug.

`GetXOPMenuID`

Use ResourceToActualMenuID instead. See **Menu Items Added By STR# 1101** for details.

`GetXOPItemID`

Use ResourceToActualItem instead. See **Menu Items Added By STR# 1101** for details.

`GetXOPSubMenuID`

Use ResourceToActualMenuID instead. See **Menu Items Added By STR# 1101** for details.

`GetXOPSubMenu`

Use ResourceMenuIDToMenuHandle instead. See **Menu Items Added By STR# 1101** for details.

`EnableXOPMenuItem`

Use EnableMenuItem instead. See **Menu Items Added By STR# 1101** for details.

```
SetXOPItem
```

Use SetMenuItemText instead. See **Menu Items Added By STR# 1101** for details.

```
DisposeXOPSubMenu
```

There is no need to dispose XOP submenus anymore. See **Menu Items Added By STR# 1101** for details.

```
DefaultMenus
```

This has been a NOP since 1992 so you can just remove it.

# New XOPSupport Routines

The following XOP Toolkit routines have been added:

```
int IsMacOSX(void)
```

Macintosh only. Returns the truth that we are running under Mac OS X.

```
void CheckItem(MenuHandle menuH,short item,int checked);
short CountMItems(MenuHandle menuH);
void DisableItem(MenuHandle menuH, short item);
void EnableItem(MenuHandle menuH, short item);
void getmenuitemtext(MenuHandle menuH, short itemNumber, char* itemString);
void setmenuitemtext(MenuHandle menuH, short itemNumber, char* itemString);
void insertmenuitem(MenuHandle menuH, char *itemString, short after);
void appendmenu(MenuHandle menuH, char *itemString);
```

These routine used to be provided by the Mac OS on Macintosh and by the XOP Toolkit on Windows. In Carbon, Apple renamed or removed them. To avoid breaking old XOPs, they now provided by the XOP Toolkit on both platforms.

```
XOPSetContextualHelpMessage(XOP_WINDOW_REF w, const char* msg, const Rect* r)
```

Allows an XOP to provide tips for controls and icons in its window. See the comment in XOPSupport.c for details.

# Menu Items Added By STR# 1101

If your XOP adds a menu item to Igor using the STR#,1101 resource, you must modify it for the Carbon XOP Toolkit. This was a deprecated technique supported only on Macintosh and is no longer supported in the Carbon XOP Toolkit. You must use the XMI1 resource instead. This is simple. Just copy the XMI1 resource from GBLoadWave.r, paste it into your .r file, and modify it for your purposes.

The following XOP routines, which supported the STR#,1101 method of adding a menu item, have been removed from the Carbon XOP Toolkit:

```
GetXOPMenuID - Use ResourceToActualMenuID instead.
GetXOPItemID - Use ResourceToActualItem instead.
GetXOPSubMenuID - Use ResourceToActualMenuID instead.
GetXOPSubMenu - Use ResourceMenuIDToMenuHandle instead.
EnableXOPMenuItem - Use EnableMenuItem instead.
SetXOPItem - Use SetMenuItemText instead.
DisposeXOPSubMenu - There is no need to dispose XOP submenus anymore.
```

Here is code that uses EnableMenuItem to enable a menu item added by an XOP to a built-in Igor menu. In this case, the item was added to Igor's Load Waves submenu by the first entry in the XOP's XMI1 resource.

```
int itemNumber = ResourceToActualItem(LOAD_SUB_ID, 1);
MenuHandle mH = GetMenuHandle(LOAD_SUB_ID);
if (mH != NULL)
   EnableMenuItem(mH, itemNumber);
```

Old XOPs that use the STR#,1101 method for adding a menu item will continue to work under Mac OS 9.

# Dialogs Under Carbon

Long ago, the Mac OS provided a set of Dialog Manager calls that you could call to manipulate dialog items. Then Apple introduced new capabilities implemented by a part of the OS called the Appearance Manager. The method for manipulating a dialog item (e.g., setting the text of an EditText field) depends on whether you use Appearance Manager features or not. All modern code uses Appearance Manager techniques.

To avoid the need to support two ways of doing the same thing, the Carbon XOP Toolkit dialog support routines *assume* that the dialog is Appearance-Manager savvy. You make your dialog Appearance savvy by:

1. Including a dlgx resource with the same resource ID as your DLOG resource.
2. Including the kDialogFlagsUseControlHierarchy bit in the dlgx resource.
3. Using Appearance Manager programming techniques for implementing your dialog. If your dialog is based on WaveMetrics examples, a large part of this is converting UserItem items to Control items, as discussed below.

The GBLoadWave XOP illustrates these points.

GBLoadWave.r includes the dlgx resource. Including the kDialogFlagsUseControlHierarchy bit in the dlgx resource causes the Mac OS to create a control handle for each item in your dialog, if the item is not already defined as a CNTL item. All UserItem controls, mostly popup menus, group boxes, underlines, and command boxes have been converted to Control items. The order of items involving group boxes has been changed.

Most, if not all of the Appearance Manager code changes needed for normal Igor-like dialogs are implemented in the XOP Toolkit dialog support routines. If you want to go beyond this, you need to learn about the Mac OS Dialog Manager, Control Manager and Appearance Manager. You will also no doubt need to study Apple's sample code. Making sense of Apple's documentation and sample code is a challenge.

## Moveable Dialog Box

You can change your dialog from immovable to movable by changing dBoxProc to movableDBoxProc in your DLOG resource.

## Default and Cancel Dialog Items

If you use the DoXOPDialog routine, you should add the following lines after the GetXOPDialog call:

```
SetDialogDefaultItem(theDialog, <Your default item number>);
SetDialogCancelItem(theDialog, <Your cancel item number>);
SetDialogTracksCursor(theDialog, 1);
```

DoXOPDialog now calls a standard Mac OS dialog filter routine that handles hits on the default and cancel items and sets the cursor. These calls let the standard Mac OS dialog filter routine know what to do.

## Macintosh Dialogs With Popup Menus

This section is of interest only if you have a pre-Carbon XOP that uses XOP Toolkit dialog popup menus.

Prior to Carbon, the XOP Toolkit implemented dialog popup menus using WaveMetrics homebrew code. This code did not fit in well with Carbon and especially with the Mac OS X Aqua look and feel. It also would have been difficult to port to Carbon. Consequently, we now use the standard Carbon methods for implementing popup menus, which means that you need to modify your code.

Most of the changes are buried inside XOPSupport routines, and you don't need to worry about them. However, there are a few things that you must do to make your XOP work with the new popup menu implementation.

In the old method, the popup menu dialog item was declared in the DITL resource as a UserItem. In the new method, the item must be a Control item. This Control item refers to a CNTL resource which you must also include in your resource fork.

Here is the old definition of the Path popup menu item in the GBLoadWave dialog:

```
resource 'DITL' (1260) {      /* Main dialog */
   {
      .
      .
      .

      /* [13] */
      {127, 55, 146, 173},
      UserItem {
         enabled
      },
```

Here is what it looks after carbonization:

```
resource 'CNTL' (1100, "Path Popup Menu", purgeable) {
   {127, 55, 146, 173},
   0,       // Title constant.
   visible,
   0,       // Width of title in pixels.
   -12345,     // MENU resource ID; MUST BE -12345!
   kControlPopupButtonProc | kControlPopupFixedWidthVariant,   // CDEF ID
   0,       // Refcon
   ""       // Title
};

resource 'DITL' (1260) {         /* Main dialog */
   {
      .
      .
      .

      /* [13] */
      {127, 55, 146, 173},
      Control {
         enabled,
         1100
      },
```

So, to convert your old XOP to use the new popup menu methods, you need to change your UserItem item into a Control item and add a corresponding CNTL resource.

Your CNTL resources should use resource IDs in the range 1100 to 1199.

Make sure that the bounds rectangle in the CNTL resource matches the bounds rectangle for the corresponding dialog item in the DITL resource.

The CNTL resource fields are nominally called initial value, visibility, maximum value and minimum value. However, when used for a popup menu, they really mean something else. The initial value field really stores something called the "title constant". The maximum field really stores the width of the title in pixels. We use 0 for these because we create an explicit title item. The minimum field really stores a MENU resource ID. This kludge is more or less explained in Apple's Control Manager documentation.

You must specify -12345 as the MENU resource ID. This prevents the Mac OS Control Manager from attempting to create a menu from a resource. The menu is created when you call the CreatePopMenu XOPSupport routine and is disposed when you call DisposeXOPDialog.

Make sure to set the bounds field of the CNTL resource to the same coordinates as the corresponding item in the DITL.

## Other Controls In Dialogs

Some other dialog items previously implemented as UserItems should be converted to controls under Carbon.

Igor command boxes in dialogs that generate commands should be converted to group box (kControlGroupBoxTextTitleProc) controls.

Group boxes should be converted to group box (kControlGroupBoxTextTitleProc) controls.

Underlines should be converted to separator (kControlSeparatorLineProc) controls.

These are all illustrated in the GBLoadWave.r file. As with popup menu items, to convert a UserItem to a control, you must add a CNTL resource and change the UserItem into a Control item, which references the CNTL resource. Make sure to set the bounds field of the CNTL resource to the same coordinates as the corresponding item in the DITL.

Pre-carbon, the group box UserItem appeared in the DITL after the items inside the group box. With carbon, this is reversed; the group box control must appear in the DITL before the items inside the group box. Also, pre-Carbon, the title for the group box was a separate dialog item. In Carbon, it is part of the group box item, so the separate title item must be removed.

# Carbon API Versus Classic Mac API

If your existing XOP calls Macintosh OS routines, you will need to make changes so that it can compile and run under Carbon. You'll know this because you will get compile errors when you try to compile using the Carbon headers.

The starting point for understanding the necessary changes is Apple's "Carbon Porting Guide", which is included with the Carbon SDK. To get you started, here is a bit of information about the changes you may need to make.

In the classic Mac API, you could directly reference fields in structures defined by the API. For example, to find the font used by the a QuickDraw GrafPort, you could write:

```
int theFont = someGrafPort->txFont;
```

In Carbon, this is not allowed. You need to call an "accessor routine", like this:

```
int theFont = GetPortTextFont(someGrafPort);
```

The names of some routines have been changed. Prior to Carbon, you could enable a menu item like this:

```
EnableItem(menuH, itemNumber);
```

In Carbon, the EnableItem routine no longer exists. You need to use EnableMenuItem instead. (But the XOP Toolkit defines EnableItem on both Macintosh and Windows because it is part of a cross-platform set of routines that pre-date Carbon.)

The classic Mac API included routine that accepted C strings as parameters. For example:

```
setitem(menuH, itemNumber, itemText);
```

where itemText is a C (null-terminated) string.

In Carbon, setitem does not exist. Neither does its more modern cousin, setitemtext. Instead, you need to use SetMenuItemText, which takes a Pascal string, not a C string. So you need to write:

```
unsigned char pItemText[256];
CopyCStringToPascal(itemText, pItemText);
SetMenuItemText(menuH, itemNumber, pItemText);
```

Some of these changes are documented in Apple's "Carbon Porting Guide". The Carbon header files also sometimes provide the necessary information. Sometimes you can find the information at Apple's Carbon documentation web site:

```
http://developer.apple.com/techpubs/macosx/Carbon/
```

Sometimes you have to search the Carbon mailing list archives:

```
http://lists.apple.com/mailman/listinfo/carbon-development
```

And sometimes you are just plain out-of-luck.

# Accessing Resources

This section discusses a problem that does not affect most XOPs. It does affect XOPs that directly or indirectly (through system or library calls) access resources.

Prior to Carbon, Igor did some sleight-of-hand to make sure that the resources in all XOPs were hidden from Igor and from all other XOPs. This trick was necessary because, way back in the days of 68K processors, the operating system could mistakenly load a CODE resource from an XOP when it should have loaded a CODE resource from Igor or from a different XOP. The trick involved setting a low-memory global which controlled the chain of resource files visible to the Resource Manager.

With Carbon, the trick is no longer feasible. In Igor Pro Carbon, all XOP resource forks are now visible to the Resource Manager all of the time. This has the potential to cause problems.

Before Igor sends a message to your XOP's XOPEntry routine, it sets the current resource fork to your XOP's resource fork. However, for speed reasons, it does not do this when calling a direct external function. Also, you may use Macintosh programming techniques, such as Carbon Events, which result in your code being called by the operating system, not by Igor. In this case, you can not be sure what resource fork is current. This means that you must take greater care when accessing resources to make sure that you access only your own resources. Here are guidelines for doing this:

1. Before calling any function that directly or indirectly accesses resources, call UseResFile to make sure that the file in which the resource resides is the current resource file. When you are finished, set the current resource file back to what it was. For example:

   ```
   int saveResFile = CurResFile();
   UseResFile(XOPRefNum());         // Set current to XOP resource fork.
   <Access resources>
   UseResFile(saveResFile);
   ```

2. Don't call routines that search multiple resource forks. For example, use Get1Resource, not GetResource. Use Get1NamedResource, not GetNamedResource.
3. There are some Mac OS routines which do not give you the option of restricting the search to one resource fork. These include GetMenu and GetIndString. This does not cause a problem if you set the current resource fork before calling these routines and if the resource is found.

However, if the resource is absent from the resource fork that you intend to search, these routines might find the resource in another resource fork.

It is possible to add a test to make sure that the resource you are trying to access is in the current resource fork. The GetXOPIndString XOPSupport routine does such a test. You should use GetXOPIndString instead of GetIndString. However, GetXOPIndString takes a pointer to an array of char and returns a C string while GetIndString takes a pointer to unsigned char and returns a Pascal string. Therefore changing from GetIndString to GetXOPIndString will require some additional work.

Prior to Carbon, the XOP Toolkit provided routines named XOPOpenResFile, XOPUseResFile and XOPCloseResFile which allowed you to open other resource files without interfering with Igor's resource hiding trick. These routines are not supported in the Carbon XOP Toolkit. If you use them, replace them with calls to OpenResFile, UseResFile and CloseResFile, and be aware of the issues discussed above in this section.

Normally, a pre-Carbon XOP can run with the Carbon version of Igor on Mac OS 9. If a pre-Carbon XOP relies on XOPOpenResFile, XOPUseResFile and XOPCloseResFile, it may or may not need to be rewritten. These routines were supplied by the XOPSupport library and did not do a callback to Igor, so it is possible that old XOPs that use them will still work with Carbon Igor. You need to try such XOPs to find out.

# Other Issues

Igor Pro Carbon includes a floating help window. This is used to replace Apple's balloon help system, which is not supported under Carbon or Mac OS X. Because the window is floating, if it is displayed, any call to FrontWindow will return the WindowRef for the floating help window. Therefore, if your XOP calls FrontWindow, you must change this to call FrontNonFloatingWindow.

# Changes That Affect Windows XOPs

The BoxItem XOPSupport routine has been removed. This was a NOP. If you use it in your XOP, you can just delete it.

The GetStandardFilePath and SetStandardFilePath routines have been removed.

*Appendix* C

# XOP Toolkit 5 Release History


Overview ........................................................ 541
Operation Starter Code Bug In Igor Pro 5.00 and 5.01 ........................................................ 541
Release 5.00, March 23, 2004 ........................................................ 541
Release 5.03, September 13, 2004 ........................................................ 541
Release 5.04 ........................................................ 542

## Overview

XOP Toolkit version numbers are chosen to roughly coincide with the version of Igor Pro shipping at the time of the XOP Toolkit release. Thus, there is an XOP Toolkit 5.00 and an XOP Toolkit 5.03 but no 5.01 or 5.02.

## Operation Starter Code Bug In Igor Pro 5.00 and 5.01

Igor Pro 5.00 and 5.01 had a bug in the automatic code generation for external operations. This bug is mostly asymptomatic because the recommended project settings force two-byte structure alignment but you should still fix it. Search your XOP source code and replace any occurrence of:

```
#pragma XOP_SET_STRUCT_PACKING
```

with

```
#include "XOPStructureAlignmentTwoByte.h"
```

Also replace any occurrence of:

```
#pragma XOP_RESET_STRUCT_PACKING
```

with

```
#include "XOPStructureAlignmentReset.h"
```

Now verify that each #include "XOPStructureAlignmentTwoByte.h" is balanced by a corresponding #include "XOPStructureAlignmentReset.h".

## Release 5.00, March 23, 2004

Initial release.

## Release 5.03, September 13, 2004

Added this routine to make it easier to call routines that expect Posix paths on Mac OS X:

```
HFSToPosixPath
```

Added the ability to pass an Igor Pro structure to an external operation. See **Structure Parameters** on page 168. This feature requires Igor Pro 5.03 or later.

Added the ability to pass an Igor Pro structure to an external function. See **Structure Parameters** on page 194. This feature requires Igor Pro 5.03 or later.

Added the following XOPSupport routines, which require Igor Pro 5.03 or later, to allow an external function to use NVAR and SVAR fields in Igor Pro structures:

```
GetNVAR

SetNVAR

GetSVAR

SetSVAR
```

# Release 5.04

Added the GetLeafName utility routine. This routine returns the leaf part of a file path.

Changed the allowable values for the fullFilePath parameter to the XOPOpenFileDialog and XOPSaveFileDialog routines. Previously these parameters were required to be either "" or just a file name. Now they can also be a full file path.

The reason for this change is that the sample XOPs incorrectly passed a full file path for this parameter. On Macintosh this was asymptomatic but on Windows it caused the wrong folder to be initially displayed in the Open File or Save File dialog when you executed, for example, SimpleLoadWave/P=<path>. The change to XOPOpenFileDialog and XOPSaveFileDialog makes acceptable the previously incorrect parameter, thus fixing the problem without requiring source code changes to all XOPs modeled on SimpleLoadWave. They still need to be relinked with the new XOPSupport library to get the benefit of this bug fix.

Added the GetTextWaveData and SetTextWaveData XOPSupport routines. GetTextWaveData allows you to get all of the data for an entire text wave in one call. SetTextWaveData allows you to set all of the data for an entire text wave in one call. These routines require Igor Pro 5.04 or later.

Added the GetWaveDimensionLabels and SetWaveDimensionLabelsXOPSupport routines. GetWaveDimensionLabels allows you to get all of the dimension labels for a wave in one call. SetTextWaveData allows you to all of the dimension labels for a wave in one call. These routines require Igor Pro 5.04 or later.

Fixed a bug in MDGetWaveScaling. Previously it could return incorrect data full scale values in rare cases.

Added the SetOperationWaveRef XOPSupport routine. If you use a DataFolderAndName parameter in an external operation to specify a destination wave, you should add a call to

SetOperationWaveRef to your ExecuteOperation function. See the documentation for SetOperationWaveRef for details. This requires Igor Pro 5.04 or later.

Using Igor Pro 5.04 or later, you can use the CheckFunctionForm and CallFunction XOPSupport routines to call a user-defined or external function that takes a structure parameter.

Added the MDChangeWave2 XOPSupport routine for redimensioning a wave without changing its data. This requires Igor Pro 5.04 or later.

Added the IsINF32 and IsINF64 XOPSupport routines. These routines work with any version of Igor.

Added the ScaleClipAndRound XOPSupport routine. This works with any version of Igor.

Added the extended form of structure parameter for external operations. This requires Igor Pro 5.04 or later. See **Extended Structure Parameters** on page 172.

# Index

## .


- .c files, 15
- .dsp files, 90
- .dsw files, 90
- .exp files
  - CodeWarrior, 77
  - Xcode, 81, 84
- .igr extension, 292
- .ihf extension, 292
- .NET, 94–98
  - Visual C++ 7, 94
- .plc files, 75
- .r files, 15
  - compiled by Rez, 109
- .rc files, 15, 109
  - Visual C++ 6, 91
  - Visual C++ 7, 96
- .rsrc files
  - in packages, 109
- .sln files, 95
- .vcproj files, 95
- .xop extension, 7, 20, 105


## _


- __initialize, 40, 72, 101
- __terminate, 40, 72, 101


## A


- aborting
  - CheckAbort, 482
- About Igor dialog, 26
- access paths
  - CodeWarrior, 70
- accessing data folders, 376–91
- accessing variables, 368–75
- accessing waves, 336–65
- ACTIVATE message, 116
  - not sent on Windows, 253
- Activity Monitor
  - leaks, 310
- ActualToResourceItem, 241, 394
- ActualToResourceMenuID, 240, 393
- adding commands, 146–78
- adding dialogs, 269–76
- adding functions, 181–204
- adding menus and menu items, 231–46
- adding operations, 146–78
- adding windows, 249–58
- AddPopMenuItems, 406
- Adobe Acrobat, 10
- afxres.h, 89
- alerts
  - custom error alert, 131
  - XOPOKAlert, 495
  - XOPOKCancelAlert, 495
  - XOPYesNoAlert, 495
  - XOPYesNoCancelAlert, 496
- aliases, 20
- alignment. (see structure alignment)
- ALRT resources, 518
- Appearance Manager, 531
- appendmenu, 502, 530
- ArrowCursor, 434
- as keyword, 155
- Asian text
  - ConcatenatePaths, 414
  - FullPathPointsToFile, 415
  - FullPathPointsToFolder, 416
  - GetDirectoryAndFileNameFromFullPath, 415
  - GetNativePath, 413
  - HFSToPosixPath, 412
  - MacToWinPath, 411
  - strchr2, 492
  - strrchr2, 492
  - WinToMacPath, 412
- AtEndOfCommand, 335


## B


- background processing
  - SpinProcess, 484
- balloon help, 295
  - for menu items, 296
  - Igor Tips, 295
  - STR# resources, 296
  - WindowXOP1 XOP example, 296
  - Xcode, 88
- big-endian
  - MDChangeWave2, 349
- BoxItem, 529
- bugs, 303–17
- bundles

Xcode, 82
byte order
MDChangeWave2, 349
byte reordering
settings, 134

C

C strings, 193, 220
C++
catch, 100
CodeWarrior, 101
exceptions, 100
mixing with C, 99
new operator, 101
templates, 214
try, 100
type coercion, 99
Visual C++, 102
WaveData, 99
writing XOPs in, 99
Xcode, 84, 102
CalcWaveRange, 165, 335
callbacks, 16, 21
IGOR.lib, 16
CallFunction, 285, 471
cancel button, 532
Cancel button, 277
cancelling
CheckAbort, 482
Capitalize, 335
Carbon
API changes, 535
balloon help, 295
dialogs, 531
FrontWindow, 537
history, 5
Igor Pro versions, 9
issues, 522
paths, 268
popup menus, 532
porting XOP to, 521
resources, 536
UserItem controls, 533, 534
XOP Toolkit 5, 5
catch, 100
categories
for functions, 185
of operations, 149
CatPossiblyQuotedName, 460
CFM, 6
creating projects in CodeWarrior, 67
resources, 109
ChangeWave, 338
chart displays, 448–49
CheckAbort, 482
CheckFunctionForm, 285, 469
CheckItem, 504, 530
CheckName, 457
CheckTerm, 335
child data folder, 381
CHUNKS, 216, 336, 360
CLEANUP message, 113
CleanupName, 457
CLEAR message, 121
CLEAR_MODIFIED message, 124, 133
CLICK message, 118
not sent on Windows, 253
CLOSE message, 117
close type code for windows, 117
clut resources, 518
CMD message, 115
error codes, 127
CmpStr, 492
CNTL resources, 275, 533
Code Fragment Manager, 6
CodeWarrior, 66–78
.exp files, 77
.mcp files, 68
.plc files, 75
__initialize, 40
__terminate, 40
access paths, 70
building SimpleFit, 30
C++, 101
CFM, 67
creating a new CFM project, 67
creating a new Mach project, 75
creating project in guided tour, 36
debugging a Macintosh XOP, 73, 78
entry points, 72
first XOP support, 4
frameworks, 75
GDB, 73
IOKit, 75
IXOP XOP file type, 39, 76
Mach-O, 75
MetroNub, 73, 78
NEWMODE, 101
prefix file, 71, 76
project files, 68
project settings, 69
shared library, 68
structure alignment, 280
supported versions, 8, 65

target settings, 69
TARGET_RT_MAC_MACHO, 77
testing installation, 11
versus Xcode, 6
XOPSupport project files, 13
colons, 267, 268
color tables
GetIgorColorTableInfo, 462
GetIgorColorTableValues, 462
GetIndexedIgorColorTableName, 461
GetNamedIgorColorTableHandle, 461
routines, 461–63
COLUMNS, 216, 336, 360
command template, 152
command templates, 155
length limit, 175
mnemonic names, 158
commands
adding, 146–78
parsing, 335
PauseUpdate, 478
PutCmdLine, 485
ResumeUpdate, 478
XOPCommand, 476
XOPCommand2, 476
XOPSilentCommand, 477
commas
in operations, 146
communicating with Igor, 326–28
compatibility, 4, 140–42
external operations, 177
Igor 1.2, 141
Igor Pro, 141
of XOPs with Igor versions, 6
operations, 147
resources, 518
XOP Toolkit 5, 507
XOPI resource, 111
compilableOp operation category, 148, 149
compiling
help files, 292
complex conjugate
example external function, 182
complex conjugate example, 189
complex numbers
external function example, 182
parameters in external functions, 189
results in external functions, 189
storage in waves, 216
ConcatenatePaths, 414
configurations
Visual C++ 6, 91
Visual C++ 7, 96
content region, 255, 256
context-sensitive help
for dialogs, 300
ConvertData, 451
ConvertData2, 452
COPY message, 121
CountMItems, 502, 530
crash logs, 319
CreatePopMenu, 245, 275, 405, 526
CreateValidDataObjectName, 458
CreateWindowEx, 251
CreateXOPWindow, 252
CreateXOPWindowClass, 252
cross-platform development, 266
dialogs, 270–76
files, 267
Macintosh emulation, 497
current data folder, 379
cursors
ArrowCursor, 434
HandCursor, 434
IBeamCursor, 434
setting, 118
setting in NULLEVENT, 118
setting on Macintosh, 250
SpinCursor, 434
WatchCursor, 434
curve fitting
SimpleFit sample XOP, 18, 29
SimpleGaussFit sample XOP, 29
WAVE_TYPE, 189
custom errors
adding, 130
CUT message, 120


# D


dangling pointers, 312
data folders, 224–28
accessing, 376–91
child, 381
commonly used XOPSupport routines, 224
conventions, 227
current, 379
DuplicateDataFolder, 383
DuplicateDataFolderObject, 389
FetchWaveFromDataFolder, 339
GetCurrentDataFolder, 379
GetDataFolder, 335
GetDataFolderAndName, 335
GetDataFolderByIDNumber, 380
GetDataFolderIDNumber, 377

GetDataFolderListing, 377
GetDataFolderNameOrPath, 376
GetDataFolderObject, 385
GetDataFolderProperties, 377
GetIndexedChildDataFolder, 381
GetIndexedDataFolderObject, 384
GetNamedDataFolder, 379
GetNumChildDataFolders, 381
GetNumDataFoldersObjects, 384
GetParentDataFolder, 381
GetRootDataFolder, 379
GetWavesDataFolder, 381
ID numbers, 377, 380
KillDataFolder, 382
KillDataFolderObject, 389
listing, 377
MoveDataFolder, 383
MoveDataFolderObject, 390
NewDataFolder, 382
Packages data folder, 227
parent, 381
RenameDataFolder, 383
RenameDataFolderObject, 391
root, 379
SetCurrentDataFolder, 379
SetDataFolderObject, 387
SetDataFolderProperties, 377
variables, 368
waves, 381
data full-scale, 341
MDGetWaveScaling, 349, 542
MDSetWaveScaling, 350
data sharing, 139
DATAFOLDER_OBJECT, 376–91
CheckName, 457
CreateValidDataObjectName, 458
DataFolderAndName parameters
external operations, 166
DataFolderHandles, 224
DataObjectValuePtr
GetDataFolderObject, 385
SetDataFolderObject, 387
dates, 494
DateToIgorDateInSeconds, 494
dBoxProc, 532
debug configuration
Visual C++ 6, 91
Visual C++ 7, 96
debugging, 318–20
crash logs, 319
Dr. Watson, 320
in CodeWarrior, 73, 78
in Visual C++ 6, 93
in Visual C++ 7, 98
in Xcode, 86
LaunchCFMApp, 86
MacsBug, 319
MetroNub, 73, 78
symbolic debugging, 318
XOPNotice, 319
default button, 532
default help file name, 293
DefaultMenus, 530
DeleteMenuItem, 502
DeleteMenuItems, 525
DeletePopMenuItems, 407
dereferencing a handle, 313
dereferencing handles, 263, 264
destination waves, 166
DestroyXOPWindow, 252
DEV_SYS_CODE, 111
Xcode, 84
development system, 111
supported systems, 65
development systems
supported systems, 8
dialogs, 398–410
adding, 269–76
AddPopMenuItems, 406
cancel button, 532
Cancel button, 277
Carbon, 531
context-sensitive help, 300
CreatePopMenu, 405
DeletePopMenuItems, 407
DialogStorage structure, 272
DisableDControl, 401
DisplayDialogCmd, 277, 404
DisposeDialogStorage, 272
DisposeXOPDialog, 399
dlgx resources, 270
DLOG and DITL resources, 269
Do It button, 277
DoXOPDialog, 398
EnableDControl, 401
FillPathPopMenu, 409
FillPopMenu, 408
FillWavePopMenu, 408
FillWindowPopMenu, 409
FinishDialogCmd, 277, 404
GetCheckBox, 400
GetDBox, 399
GetDDouble, 403
GetDInt, 402
GetDLong, 402
GetDText, 401

GetPopMenu, 407
GetPopMenuHandle, 406
GetRadBut, 400
GetXOPDialog, 398
HandleItemHit, 272
help, 300
Help button, 277
HiliteDControl, 401
Igor style, 276
IgorError, 483
InitDialogSettings, 272
InitDialogStorage, 272
InitPopMenus, 405
ItemIsPopMenu, 406
kDialogFlagsUseControlHierarchy, 270
KillPopMenus, 410
on Macintosh, 270
on Windows, 270
popup menus, 245, 275, 277
resources, 269
SelEditItem, 403
SelMacEditItem, 403
SetCheckBox, 400
SetDDouble, 403
SetDialogBalloonHelpID, 399
SetDialogPort, 398
SetDInt, 402
SetDLong, 402
SetDText, 401
SetPopItem, 407
SetPopMatch, 407
SetRadBut, 400
ShowDialogWindow, 398
ShutdownDialogSettings, 272
To Clip button, 277
To Cmd button, 277
ToggleCheckBox, 400
UserItem controls, 533, 534
XOP_WINDOW_REF, 270
XOPDialog, 399
XOPDisplayHelpTopic, 486
XOPOpenFileDialog, 421
XOPSaveFileDialog, 423
DialogStorage structure, 272
dimension labels
GetWaveDimensionLabels, 353, 542
MDGetDimensionLabel, 352
MDSetDimensionLabel, 353
SetWaveDimensionLabels, 354, 542
dimension scaling
for waves, 218
MDGetWaveScaling, 349
MDSetWaveScaling, 350
dimensions
MDChangeWave, 348
MDChangeWave2, 349
MDGetWaveDimensions, 347
MDMakeWave, 346
ROWS, COLUMNS, LAYERS, CHUNKS, 216, 336
direct access method, 214
direct functions, 201
direct method for external functions, 201–2
directory IDs, 267
DisableDControl, 401
DisableItem, 503, 530
disabling menu items, 242–44
DisplayDialogCmd, 277, 404
DISPLAYSELECTION message, 122
DisposeDialogStorage, 272
DisposeHandle, 499
DisposeMenu, 245
DisposePtr, 498
DisposeWindow, 250
DisposeXOPDialog, 399
DisposeXOPSubMenu, 530
DITL resources, 269
dlgx resources, 270, 531
DLOG resources, 269, 518
Do It button, 277
DoCHIO, 483
DoFunction routine, 201
DOITID, 404
DOUBLE, 518
DoUpdate, 478
recursion, 137
DoWindowRecreationDialog, 466
DoXOPDialog, 398
Dr. Watson, 320
DUPLICATE message, 122
DuplicateDataFolder, 383
DuplicateDataFolderObject, 389

# E

EditItem, 527
email, 26
EnableDControl, 401
EnableItem, 503, 530
EnableXOPMenuItem, 529
enabling menu items, 242–44
endian
MDChangeWave2, 349
entry points
CodeWarrior, 72
error codes

from operations, 147
errors, 127–31
custom error alert, 131
custom errors, 130
custom XOP error codes, 128
error codes, 127
external functions, 187
FIRST_XOP_ERR, 130
GetIgorErrorMessage, 127, 131, 483
GetLastError, 128–30
Igor error codes, 127
IgorError, 127, 131, 483
in external functions, 202
Mac OS error codes, 128
STR# 1100 resource, 130
Windows OS error codes, 128–30
WindowsErrorToIgorError, 128–30, 473, 474
WMGetLastError, 128–30
XOPOKAlert, 131
XOPOKCancelAlert, 131
XOPYesNoAlert, 131
XOPYesNoCancelAlert, 131
exceptions
in C++, 100
ExecuteOperation, 152
NULL handles, 175
EXP_xxx, 125, 126
experiment type argument, 125, 126
experiments, 124–26, 132–34
CLEAR_MODIFIED message, 124
EXP_xxx, 125, 126
LOAD message, 124
LOAD_TYPE_xxx, 126
loading settings, 133
LOADSETTINGS message, 126
MODIFIED message, 124
NEW message, 124
SAVE message, 124
SAVE_TYPE_xxx, 125
SAVESETTINGS message, 125
saving settings, 133
EXPORT_GRAPHICS message, 122
Exports.exp file, 43
extended structure parameters, 172
example in external operation, 173
extensions
help files, 292
extern declaration
mixing C and C++, 99
external functions, 181–204
adding, 182
calling from an XOP, 285
complex parameters, 189
complex results, 189
direct functions, 201
direct method, 201–2
DoFunction routine, 201
error codes, 187, 202
examples, 182
FUNCADDRS message, 114, 201–2
function categories, 185
FUNCTION message, 115, 201–2
FV_REF_TYPE, 199
guided tour, 29
invoking from Igor, 186
keep RESIDENT, 201
logfit example, 202
message method, 201–2
names of, 183, 186
parameter types, 183, 188
parameters, 181, 187
pass-by-reference, 198
pass-by-value, 198
result types, 183, 188
returning results, 187
SimpleFit sample XOP, 18
string parameters, 190, 191, 193, 199
string results, 190, 191
structure alignment, 187
structure parameter example, 196
structure parameters, 194, 195
versus user functions, 181
wave assignment statements, 186
wave parameters, 204
WaveAccess sample XOP, 19
XFUNC1 sample XOP, 17
XFUNC2 sample XOP, 17
XFUNC3 sample XOP, 17
external operations, 146–78
as keyword, 155
command template, 152
command templates, 155
compatibility, 177
compatibility with Igor Pro 4, 147
compilableOp, 148
compilableOp operation category, 149
DataFolderAndName parameters, 166
extended structure parameter example, 173
extended structure parameters, 172
flags, 146, 154
GBLoadWave sample XOP, 18
keywords, 154
MenuXOP1 sample XOP, 18
mnemonic names, 158
name parameters, 163
names of, 150

NIGPIB2 sample XOP, 19
NVARs, 177
Operation Handler, 151–78
Operation Handler XOPSupport routines, 329
optional parameters, 157, 162
output variables, 174
parameters, 146, 154
RegisterOperation, 329
runtime parameter structure, 159
SimpleLoadWave sample XOP, 18
starter code, 152
starter code bug, 541
string parameters, 163
structure parameter example, 169
structure parameters, 168
template, 152
templates, 155
TUDemo sample XOP, 19
variables, 330, 331, 332, 333, 428
VarName parameters, 165
VDT2 sample XOP, 19
wave parameters, 164
wave references, 333
WaveRange parameters, 164
WindowXOP1 sample XOP, 17
XOP1 sample XOP, 17
XOPOp operation category, 149

# F

F_EXTERNAL function category, 185
FetchNumericValue, 362
FetchNumVar, 369
    example, 223
FetchStrHandle, 370
FetchStrVar, 370
    example, 223
FetchWave, 338
FetchWaveFromDataFolder, 339
fields
    of structure parameters, 282
FIFOs, 448–49
    GetNamedFIFO, 449
    MarkFIFOUpdated, 449
    NamedFIFO.h, 448
    SoundInput XOP, 448
file loaders
    FileLoaderGetOperationFlags, 335
    FileLoaderGetOperationFlags2, 335
    FileLoaderMakeWave, 427
    GBLoadWave sample XOP, 18
    GetFullPathFromSymbolicPathAndFilePath, 426
    Load Waves submenu, 18
    output variables, 174
    SanitizeWaveName, 456
    SetFileLoaderOperationOutputVariables, 428
    SetFileLoaderOutputVariables, 174, 428
    SetOperationFileLoaderOutputVariables, 174
    SimpleLoadWave sample XOP, 18
    support routines, 426–29
    XOPOpenFileDialog, 421
    XOPSaveFileDialog, 423
FILE_LOADER flags, 335, 427
FileFullySpecified, 528
FileLoaderGetOperationFlags, 335, 526
FileLoaderGetOperationFlags2, 335
FileLoaderMakeWave, 427
files
    accessing, 411–25
    ConcatenatePaths, 414
    cross-platform routines, 267
    directory IDs, 267
    FullPathPointsToFile, 415
    FullPathPointsToFolder, 416
    GetDirectoryAndFileNameFromFullPath, 415
    GetFullMacPathToDirectory, 416
    GetLeafName, 416
    GetNativePath, 413
    HFSToPosixPath, 412
    MacToWinPath, 411
    MAX_DIRNAME_LEN, 267
    MAX_FILENAME_LEN, 267
    MAX_PATH_LEN, 267
    MAX_VOLUMENAME_LEN, 267
    path conversions, 268
    path separator characters, 267
    platform independence, 18
    TUSFInsertFile, 441
    TUSFWriteFile, 441
    volume reference numbers, 267
    WinToMacPath, 412
    working directory refNums, 267
    XOPAtEndOfFile, 420
    XOPCloseFile, 418
    XOPCreateFile, 417
    XOPDeleteFile, 417
    XOPGetFilePosition, 420
    XOPNumberOfBytesInFile, 420
    XOPOpenFile, 417
    XOPReadFile, 418
    XOPReadFile2, 418
    XOPReadLine, 419
    XOPSetFilePosition, 420
    XOPWriteFile, 419
FillMenu, 394

FillMenuNoMeta, 395
FillPathMenu, 396
FillPathPopMenu, 409
FillPopMenu, 408
FillWaveMenu, 395
FillWavePopMenu, 408
FillWindowPopMenu, 409
FillWinMenu, 396
find
    menu item, 243, 392
    TUFind, 440
FIND message, 121, 243
FinishDialogCmd, 277, 404
FIRST_IGOR5_ERR, 127
FIRST_XOP_ERR, 128, 130
FIRSTCMD, 485
FIRSTCMDCRHIT, 485
FixByteOrder, 454
flags, 146, 154
    optional parameters, 157
    prefix characters, 155
fopen function
    Xcode, 88
forward slashes, 267, 268
frameworks
    CodeWarrior, 75
    Xcode, 83
FrontWindow, 525, 537
FTP, 8, 26
    XOP Toolkit updates, 65
full size position message, 119
FullPathPointsToFile, 415
FullPathPointsToFolder, 416
FUNCADDRS message, 114, 201–2
FUNCREFs
    in structure parameters, 282
    using from an XOP, 285
function categories, 185
    F_EXTERNAL, 185
function index, 115, 201
FUNCTION message, 115, 201–2
    error codes, 127
functions. (see also external functions)
    adding, 181–204
    CallFunction, 285
    calling from an XOP, 285
    characteristics of, 3
    CheckFunctionForm, 285, 469
    GetFunctionInfo, 285, 467, 471
    GetFunctionInfoFromFuncRef, 285, 468
FV_REF_TYPE, 199
    in XOPF resource, 184
FV_STRUCT_TYPE, 195
    in XOPF resource, 184

## G

GBLoadWave sample XOP, 18
GBLoadWave XOP
    Igor-style dialog, 276
GDB, 73
GENERATINGPOWERPC, 524
GET_TARGET_WINDOW_NAME message, 123
GET_TARGET_WINDOW_REF message, 123
GetA4, 527
GetActiveWindowRef, 430, 525
GetAString, 335
GetAStringInHandle, 335
GetCheckBox, 400
GetCStringFromHandle, 217, 220, 491
GetCurrentDataFolder, 379
GetDataFolder, 335
GetDataFolderAndName, 335
GetDataFolderByIDNumber, 380
GetDataFolderIDNumber, 377
GetDataFolderListing, 377
GetDataFolderNameOrPath, 376
GetDataFolderObject, 385
GetDataFolderProperties, 377
GetDBox, 399
GetDDouble, 403
GetDInt, 402
GetDirectoryAndFileNameFromFullPath, 415
GetDLong, 402
GetDText, 401
GetFlag, 335
GetFlagNum, 335
GetFormat, 335
GetFullMacPathToDirectory, 416, 526
GetFullPathFromSymbolicPathAndFilePath, 426
GetFunctionInfo, 285, 467
GetFunctionInfoFromFuncRef, 285, 468
GetHandleSize, 193, 220, 499
    use with strings, 321
GetIgorColorTableInfo, 462
GetIgorColorTableValues, 462
GetIgorErrorMessage, 127, 131, 483
GetIgorProcedure, 465
GetIgorProcedureList, 464
GetIndexedChildDataFolder, 381
GetIndexedDataFolderObject, 384
GetIndexedIgorColorTableName, 461
GetIndString, 537
GetKeyword, 335
GetLastError, 128–30

GetLeafName, 416
GetLong, 335
GetMenu, 245
GetMenuHandle, 240
getmenuitemtext, 503, 530
GetName, 335
GetNamedDataFolder, 379
    example, 227
GetNamedFIFO, 449
GetNamedIgorColorTableHandle, 461
GetNativePath, 268, 413
GetNewWindow, 250
GetNum, 335
GetNum2, 335
GetNumChildDataFolders, 381
GetNumDataFoldersObjects, 384
GetNumVarName, 335
GetNVAR, 374
    example, 283
GetParentDataFolder, 381
GetPath, 335
GetPathInfo2, 481
GetPopMenu, 275, 407
GetPopMenuHandle, 406
GetPrefsState, 132, 490
GetPtrSize, 497
GetRadBut, 400
GetResource, 288, 536
GetRootDataFolder, 379
    example, 227
GetStandardFilePath, 528
GetStrVarName, 335
GetSVAR, 375
    example, 283
GetSymb, 335
GetSymbolicPath, 527
GetTextWaveData, 218, 365, 542
GetTrueOrFalseFlag, 335
GetVRefNumAndDirIDFromFullPath, 528
GetWave, 335
GetWaveDimensionLabels, 353, 542
GetWaveList, 335
GetWaveName, 335
GetWaveRange, 335
GetWavesDataFolder, 381
GetWavesInfo, 344
GetXOPDialog, 398
GetXOPIndString, 447, 537
GetXOPItem, 107, 112, 328
    items field in IORecHandle, 136
    pitfalls, 320
GetXOPItemID, 529
GetXOPMenuID, 529
GetXOPMessage, 107, 112, 327
    message field in IORecHandle, 136
    pitfalls, 320
GetXOPNamedResource, 447
GetXOPPrefsHandle, 132, 490
GetXOPRefCon, 328
GetXOPResource, 447
GetXOPResult, 327
GetXOPStatus, 241, 328
GetXOPSubMenu, 529
GetXOPSubMenuID, 529
GetXOPWindow, 250, 430
GetXOPWindowIgorPositionAndState, 433
GetXOPWindowPositionAndState, 431
global variables, 303
GPIB communications
    NIGPIB2 sample XOP, 19
GRAF_MASK, 489
GROW message, 116
    not sent on Windows, 253

# H

HandAndHand, 499
HandCursor, 434
HandleItemHit, 272
handles, 139, 262, 261–65
    containing strings, 321
    dereferencing, 263, 264, 313
    heap scramble, 312
    locking, 264, 315
    master pointers, 262
    MoveLockHandle, 317
    recommended practices, 317
    treating as C strings, 193, 220
    unlocking, 316
    usage, 264
Handles
    disposing, 163
    GetCStringFromHandle, 217, 220
    PutCStringInHandle, 217, 220
HandToHand, 498
has68KHardwareFPU, 527
hasFPU, 527
hdlg resources
    Mach-O, 295
header files, 13
heap, 262
    fragmentation, 264
    overwriting, 305
heap scramble testing, 317
heaps

fragmentation, 315
heap scramble, 312
heap scramble testing, 317
help, 291–300
balloon help, 295
balloon help for menu items, 296
command help, 60
context-sensitive help, 300
example, 56
for dialogs and windows, 300
for operations and functions, 292–94
function categories, 185
help file name, 293
Igor Help Browser, 292
Igor Pro version, 34
operation categories, 149
searching for help files, 486
status line help, 297–99
technical support, 26
tips, 488
topics and subtopics, 486
XOPDisplayHelpTopic, 300, 486
XOPSetContextualHelpMessage, 488
Help Browser, 60
Help button, 277
in dialogs, 277
help files
compiling, 292
extensions, 292
location of, 292
Xcode, 88
HFSToPosixPath, 412
HGetState, 500
HideAndDeactivateXOPWindow, 431
HiliteDControl, 401
history
XOPNotice, 478
XOPResNotice, 479
HLock, 264, 315, 500
HMNU resource
status line help, 297
hmnu resources
Mach-O, 295
HMNU resources
status line help, 297
HMODULE, 473
HSetState, 264, 316, 500
HSTRING_TYPE, 191, 199
in XOPF resource, 184
HUnlock, 264, 316, 500
HWND, 116
IgorClientHWND, 473
XOP_WINDOW_REFs, 249

# I

IBeamCursor, 434
icon
Xcode, 85
ID numbers for data folders, 377, 380
IDLE message, 112
IDLES bit, 112, 136
SetXOPType, 326
idling, 135
IEEE_FLOAT, 450
Igor 1.2
compatibility, 141
Igor 1.x, 4
Igor commands, 476, 477, 485
Igor Extensions folder, 4, 9, 20, 147, 182
Igor Help Browser, 292
Igor mailing list, 26
Igor number type codes, 450, 452, 453
Igor Pro
Carbon versus pre-Carbon, 9
compatibility, 141
first Windows version, 5
history, 4
Igor Extensions folder, 9
Igor Pro 2.0, 4
Igor Pro 3.0, 5
Igor Pro 3.1, 5
Igor Pro 4, 5
Igor Pro 5 Windows compatibility, 5
Igor Pro 4
external operations, 147
pass-by-reference, 198
Igor Tips, 295
XOPSetContextualHelpMessage, 488
Igor version, 26
Igor window coordinates, 255
IGOR.lib
callback routines, 16
link with, 320
memory management routines, 261, 497
menu manager routines, 232
XOPSupport, 13
IGOR_OBSOLETE, 140, 141
IgorClientHWND, 473
IgorColorSpec, 462
IgorDateInSecondsToDate, 494
IgorError, 127, 131, 483
IgorModule, 473
Igor-style dialogs, 276
IgorVersion, 486
igorVersion global, 140

IgorXOP.h, 13
IgorXOPs5 folder
  contents of, 13
  creating, 10
IGR0 XOP file creator, 20, 105
  Xcode, 84
INDENTLEFT message, 121
INDENTRIGHT message, 121
info.plist
  Xcode, 83
INFs, 493
INIT message, 22, 112
  error codes, 127
InitDialogSettings, 272
InitDialogStorage, 272
initialization, 20, 22, 112, 288, 326
InitPopMenus, 275, 405
INSERTCMD, 485
INSERTFILE message, 122
insertmenuitem, 502, 530
installing XOP Toolkit, 10
Intel, 5
Interface Builder, 109, 270
IOKit
  CodeWarrior, 75
IORecHandle, 20, 105, 107, 135–36
  passed to main, 24
  recursion, 138
IsINF32, 493, 543
IsINF64, 493, 543
IsMacOSX, 488, 530
IsNaN32, 493
IsNaN64, 493
IsStringExpression, 335
IsXOPWindowActive, 430
ITEM_REQUIRES bits, 242
itemFlags field, 242
ItemIsPopMenu, 406
IXOP XOP file type, 20, 39, 44, 76, 105
  Xcode, 84

# K

kDialogFlagsUseControlHierarchy, 270, 531
KEY message, 118
  not sent on Windows, 253
Keyword, 335
keywords, 154
  in operations, 146
  optional parameters, 157
  parsing, 335
  prefix characters, 155
KillDataFolder, 382
KillDataFolderObject, 389
KillPopMenus, 275, 410
KillWave, 338
kMDWaveAccessMode0, 214, 355

# L

LaunchCFMApp, 86
  Xcode, 83
LAYERS, 216, 336, 360
leaks, 310
Legendre polynomials, 17, 182
liberal names, 455
  CatPossiblyQuoteName, 460
  CleanupName, 457
  CreateValidDataObjectName, 458
  GetDataFolderAndName, 335
  PossiblyQuoteName, 460
libraries
  link errors, 320
  XOPSupport, 13
linking
  errors, 320
LITTLE_ENDIAN, 518
little-endian
  MDChangeWave2, 349
LOAD message, 124, 132
Load Waves submenu, 18
LOAD_TYPE_xxx, 126
loading settings, 133
LOADSETTINGS message, 126, 133
LoadXOPSegs, 524
lock state
  waves, 340
locking handles, 264, 315
logfit
  example external function, 17, 182, 202

# M

Mac OS X
  CFM versus Mach-O, 6
  first Igor version, 5
Mach-O, 6
  creating projects in CodeWarrior, 75
  creating projects in Xcode, 80, 83
  resources, 109
Macintosh
  Igor Extensions folder, 9
  supported OS versions, 5

XOP Toolkit 5 requirements, 5
Macintosh emulation, 6
appendmenu, 502
CheckItem, 504
CountMItems, 502
DeleteMenuItem, 502
DisableItem, 503
DisposeHandle, 499
DisposePtr, 498
EnableItem, 503
GetHandleSize, 499
getmenuitemtext, 503
GetPtrSize, 497
HandAndHand, 499
HandToHand, 498
HGetState, 500
HLock, 500
HSetState, 500
HUnlock, 500
insertmenuitem, 502
MemError, 501
memory management, 139, 261
memory management routines, 497–501
menu management routines, 502–4
MoveHHi, 501
NewHandle, 498
NewPtr, 497
PtrAndHand, 501
PtrToHand, 501
routines, 497–504
SetHandleSize, 499
setmenuitemtext, 503
SetPtrSize, 498
TickCount, 504
Macintosh menu manager routines, 232
MacRectToWinRect, 496
Macsbug
heap scramble testing, 317
MacsBug, 319
MacToWinPath, 268, 411
mailing list, 26
main, 21
receives INIT message, 112
receives IORecHandle, 24
version checking, 140
main function, 105
Xcode, 84, 102
MAIN_NAME_SPACE, 455
MakeWave, 337
MallocDebug
leaks, 310
MarkFIFOUpdated, 449
master pointers, 262
MAX_DIM_LABEL_CHARS, 352, 353, 354
MAX_DIRNAME_LEN, 267
MAX_FILENAME_LEN, 267
MAX_PATH_LEN, 267
MAX_UNIT_CHARS, 342, 351, 352
MAX_VOLUMENAME_LEN, 267
MDAccessNumericWaveData, 214, 355
MDChangeWave, 348
MDChangeWave2, 349, 543
MDGetDimensionLabel, 352
MDGetDPDataFromNumericWave, 213, 359
MDGetNumericWavePointValue, 212, 357
MDGetTextWavePointValue, 363
example, 217
MDGetWaveDimensions, 216, 347
MDGetWaveScaling, 349, 542
MDGetWaveUnits, 351
MDI
child windows, 251, 252
destroying a window, 252
getting Igor client HWND, 473
getting window position, 431
menu bar, 246, 253
MOVE_TO_FULL_POSITION message, 119
RETRIEVE message, 120
setting window position, 255, 431
window coordinates, 432
MDMakeWave, 346
example, 210
text wave example, 140
MDSetDimensionLabel, 353
MDSetNumericWavePointValue, 212, 358
MDSetTextWavePointValue, 364
example, 218
MDSetWaveScaling, 218, 350
example, 210
MDSetWaveUnits, 218, 352
MDStoreDPDataInNumericWave, 213, 360
MemClear, 491
MemError, 501
memory
checking allocations, 311
dangling pointers, 312
data sharing, 139
DisposeHandle, 499
DisposePtr, 498
disposing, 309, 311
GetHandleSize, 499
GetPtrSize, 497
HandAndHand, 499
handles, 262, 261–65
HandToHand, 498
heap, 262

heap scramble, 312
HGetState, 500
HLock, 500
HSetState, 500
HUnlock, 500
leaks, 310
Macintosh emulation, 139, 261
Macintosh memory management, 261–65
MemError, 501
MoveHHi, 501
NewHandle, 498
NewPtr, 497
pointers, 262
PtrAndHand, 501
PtrToHand, 501
relocatable block, 263, 264
SetHandleSize, 499
SetPtrSize, 498
using unallocated blocks, 308
memory leaks
new operator, 101
memory management routines, 497–501
IGOR.lib, 497
menu bars, 246
menu handles, 240
menu IDs, 238, 239–40
conflicts, 245
ranges, 245
menu management routines, 502–4
menu manager routines, 232
MENU resources
for popup menus, 275
MENUENABLE message, 113, 242, 250
MenuHandles, 232
MENUHELP resources
status line help, 297
MENUITEM message, 113, 239
error codes, 127
on Windows, 253
recursion, 137
menus
ActualToResourceItem, 241, 394
ActualToResourceMenuID, 240, 393
adding, 231–46
adding a main menu, 235
adding a main menu item, 234–38
appendmenu, 502
balloon help, 296
CheckItem, 504
CountMItems, 502
DeleteMenuItem, 502
dialog popup menus, 245, 275
DisableItem, 503
disabling, 242–44
DisposeMenu, 245
EnableItem, 503
enabling, 242–44
enabling for XOP window, 250
enabling Igor items, 243
FillMenu, 394
FillMenuNoMeta, 395
FillPathMenu, 396
FillWaveMenu, 395
FillWinMenu, 396
Find item, 243, 392
GetMenu, 245
getmenuitemtext, 503
getting menu handles, 240
insertmenuitem, 502
limitations, 234
menu bars, 246
menu ID conflicts, 245
menu ID ranges, 245
menu IDs, 238, 239–40
menu manager routines, 232
MENUENABLE message, 113
MenuHandles, 232
MENUITEM message, 113, 239
MenuXOP1 sample XOP, 18, 231, 234
newmenu, 245
popup menus, 277
ReleaseMenu, 245
resource IDs, 238
ResourceMenuIDToMenuHandle, 240, 394
resources, 234–38
ResourceToActualItem, 241, 393
ResourceToActualMenuID, 240, 393
responding to, 239–41
SetIgorMenuItem, 392
setmenuitemtext, 503
SHOW_MENU_AT_LAUNCH, 235
SHOW_MENU_WHEN_ACTIVE, 235
status line help, 297–99
STR# 1101 resource, 236
TUFixEditMenu, 441
TUFixFileMenu, 441
Undo item, 392
WM_COMMAND message, 239
WM_INITMENU message, 242
WMDeleteMenuItems, 394
XMI1 1100 resource, 234, 236
XMN1 1100 resource, 234, 235
XOPSupport routines, 392–97
XSM1 1100 resource, 234, 237
MenuXOP1 sample XOP, 18
MenuXOP1 XOP, 231, 234

message boxes. (see alerts)
message method for external functions, 201–2
messages
ACTIVATE message, 116
basic, 112–14
between Igor and XOP, 136
CLEANUP message, 113
CLEAR message, 121
CLEAR_MODIFIED message, 124
CLICK message, 118
CLOSE message, 117
CMD message, 115
COPY message, 121
CUT message, 120
DISPLAYSELECTION message, 122
DUPLICATE message, 122
EXPORT_GRAPHICS message, 122
FIND message, 121, 243
for XOPs with windows, 116–23
from Igor to XOP, 20
FUNCADDRS message, 114, 201
FUNCTION message, 115
GET_TARGET_WINDOW_NAME message, 123
GET_TARGET_WINDOW_REF message, 123
GROW message, 116
IDLE message, 112
INDENTLEFT message, 121
INDENTRIGHT message, 121
INIT message, 22, 112
INSERTFILE message, 122
KEY message, 118
LOAD message, 124, 132
LOADSETTINGS message, 126, 133
MENUENABLE message, 113, 250
MENUITEM message, 113, 239
MODIFIED message, 124
MOVE_TO_FULL_POSITION message, 119
MOVE_TO_PREFERRED_POSITION message, 119
NEW message, 124, 132
NULLEVENT message, 118, 250
OBJINUSE message, 114
PAGESETUP message, 122
PASTE message, 121
PRINT message, 122
recursion, 138
REPLACE message, 121
RETRIEVE message, 120
REVERT_WINDOW message, 123
SAVE message, 124, 132
SAVE_WINDOW message, 123
SAVE_WINDOW_AS message, 123
SAVE_WINDOW_COPY message, 123
SAVE_WINDOW_MACRO message, 123
SAVEFILE message, 122
SAVESETTINGS message, 125, 133
saving and loading settings, 124–26
SELECT_ALL message, 122
SET_TARGET_WINDOW_NAME message, 123
SET_TARGET_WINDOW_TITLE message, 123
SETGROW message, 117
UNDO message, 121
UPDATE message, 116, 250
WINDOW_MOVED message, 118
XOPEntry routine, 21
meta-characters, 395, 396
FillMenu, 394
FillMenuNoMeta, 395
MetroNub, 73, 78
Metrowerks. (see CodeWarrior)
MFC, 89
Microsoft, 5
miscellaneous routines, 476–90
mnemonic names, 158
modification count
waves, 340
modification date
waves, 339
modification state
waves, 340
MODIFIED message, 124
Motorola, 4
movableDBoxProc, 532
MOVE_TO_FULL_POSITION message, 119
MOVE_TO_PREFERRED_POSITION message, 119
MoveDataFolder, 383
MoveDataFolderObject, 390
MoveHHi, 315, 501
MoveLockHandle, 262, 264, 317, 495
MPW, 4
Multiple Document Interface. (see MDI)
MW_MASK, 489

# N

name parameters
external operations, 163
NamedFIFO.h, 448
names
CatPossiblyQuotedName, 460
CheckName, 457
CleanupName, 457
CreateValidDataObjectName, 458
GetWaveName, 335
liberal, 335, 457, 458, 460
of external functions, 186

of external operations, 150
of XOPs, 7
PossiblyQuoteName, 460
SanitizeWaveName, 456
UniqueName, 455
UniqueName2, 455
WaveName, 343
NaNs, 493
National Instruments
NIGPIB2 sample XOP, 19
native paths, 268
NEW message, 124, 132
new operator
in CodeWarrior, 101
NewDataFolder, 382
example, 227
NewHandle, 262, 498
example, 263
newmenu, 245
NEWMODE
new operator, 101
NewPtr, 262, 497
example, 263
NextSymb, 335
nibs, 109, 270
NIGPIB2 sample XOP, 19
NT_CMPLX, 189
in XOPF resource, 184
number type for waves, 211
NT_FP32, 337, 450
number type for waves, 211
NT_FP64, 337, 450
in XOPF resource, 184
number type for waves, 211
NT_I16, 337, 450
number type for waves, 211
NT_I32, 337, 450
number type for waves, 211
NT_I8, 337, 450
number type for waves, 211
NT_UNSIGNED, 337, 450
number type for waves, 211
NULLEVENT message, 118, 250
not sent on Windows, 253
number type codes, 450, 452
number types
for waves, 211
GBLoadWave example, 18
NumBytesAndFormatToNumType, 453
numeric conversion
ConvertData, 451
ConvertData2, 452
FixByteOrder, 454
NumBytesAndFormatToNumType, 453
NumTypeToNumBytesAndFormat, 452
ScaleClipAndRoundData, 453
ScaleData, 453
numeric conversion routines, 450–54
GBLoadWave example, 18
numeric expressions
in operations, 146
numeric precision
in external functions, 188
NumTypeToNumBytesAndFormat, 452
NVAR, 165
NVARs
external operations, 177
GetNVAR, 374
in structure parameters, 282, 283
SetNVAR, 374

# O

object name routines, 455–60
object types
WAVE_OBJECT, 114
objects
counting in data folders, 384
duplicating in data folders, 389
getting in data folders, 384, 385
killing in data folders, 389, 390
renaming in data folders, 391
setting in data folders, 387
OBJINUSE message, 114
OpenFileReadOnly, 528
operation categories, 149
compilableOp, 149
XOPOp, 149
Operation Handler, 151–78
bug in Igor Pro 5.00 and 5.01, 541
introduced in Igor Pro 5, 5
RegisterOperation, 329
XOPSupport routines, 329
operation index, 115
operations. (see also external operations)
adding, 146–78
characteristics of, 3
optional parameters, 157, 162
output variables, 174

# P

packaged bundles
extension, 7

# Index


packages
  .rsrc files, 109
  Xcode, 81, 85
Packages data folder, 227
PAGESETUP message, 122
PANEL_MASK, 489
parameter types
  in external functions, 188
  numeric precision, 188
parameters, 154
  as keyword, 155
  checking string parameters in external functions, 193
  commas between parameters, 146
  complex in external functions, 189
  DataFolderAndName parameters, 166
  disposing string parameters in external functions, 193
  external functions, 187
  FV_REF_TYPE, 199
  in external functions, 181, 183
  in operations, 146
  name parameters, 163
  optional, 157
  optional parameters, 162
  pass-by-reference, 198
  pass-by-value, 198
  runtime parameter structure, 159
  string in external functions, 190, 191, 199
  string parameters, 163
  structures, 281
  structures in external functions, 194, 195
  structures in external operations, 168
  type checking in external functions, 184
  VarName parameters, 165
  wave in external functions, 204
  wave parameters, 164
  WaveRange parameters, 164
parent data folder, 381
ParseOperationTemplate, 152
parsing commands, 335
  AtEndOfCommand, 335
  CalcWaveRange, 335
  Capitalize, 335
  CheckTerm, 335
  FileLoaderGetOperationFlags, 335
  FileLoaderGetOperationFlags2, 335
  GetAString, 335
  GetAStringInHandle, 335
  GetDataFolder, 335
  GetDataFolderAndName, 335
  GetFlag, 335
  GetFlagNum, 335
  GetFormat, 335
  GetFullPathFromSymbolicPathAndFilePath, 426
  GetKeyword, 335
  GetLong, 335
  GetName, 335
  GetNum, 335
  GetNum2, 335
  GetNumVarName, 335
  GetPath, 335
  GetStrVarName, 335
  GetSymb, 335
  GetTrueOrFalseFlag, 335
  GetWave, 335
  GetWaveList, 335
  GetWaveName, 335
  GetWaveRange, 335
  IsStringExpression, 335
  Keyword, 335
  NextSymb, 335
pass-by-reference, 198
  FV_REF_TYPE, 199
pass-by-value, 198
PASTE message, 121
path separator characters, 267
PathList, 480
PathNameFromDirID, 416
PathNameFromDirWD, 416
paths. (see also symbolic paths)
  ConcatenatePaths, 414
  escape characters, 268
  FullPathPointsToFile, 415
  FullPathPointsToFolder, 416
  GetDirectoryAndFileNameFromFullPath, 415
  GetFullMacPathToDirectory, 416
  GetFullPathFromSymbolicPathAndFilePath, 426
  GetLeafName, 416
  GetNativePath, 268, 413
  HFSToPosixPath, 412
  Mac/Win conversion, 268
  MacToWinPath, 268, 411
  native paths, 267, 268
  POSIX paths, 267, 268
  WinToMacPath, 268, 412
PauseUpdate, 478
PEF, 6
phone number for technical support, 26
PICT resources, 518
pitfalls, 320–22
PkgInfo file
  Xcode, 85
PL_MASK, 489
platform-independence, 6
platform-independent development. (see cross-platform development)
plgndr

example external function, 17, 182
point access method, 212
pointers, 262
dangling pointers, 312
heap scramble, 312
misconceptions, 307
PopMenu, 527
popup menus, 277
AddPopMenuItems, 406
CreatePopMenu, 245, 275, 405
DeletePopMenuItems, 407
enabling and disabling, 276
FillMenu, 394
FillMenuNoMeta, 395
FillPathMenu, 396
FillPathPopMenu, 409
FillPopMenu, 408
FillWaveMenu, 395
FillWavePopMenu, 408
FillWindowPopMenu, 409
FillWinMenu, 396
GetPopMenu, 275, 407
GetPopMenuHandle, 406
in Carbon, 532
in dialogs, 275
InitPopMenus, 275, 405
ItemIsPopMenu, 406
KillPopMenus, 275, 410
Macintosh, 275
SetPopItem, 407
SetPopMatch, 407
WMDeleteMenuItems, 394
POSIX, 267, 268
POSIX paths
Xcode, 88
PossiblyQuoteName, 460
Power Macintosh, 4
PowerPC, 4
PPC, 4
preferences, 132
GetPrefsState, 490
GetXOPPrefsHandle, 490
SaveXOPPrefsHandle, 489
structures, 132
prefix characters, 155
prefix file, 76
CodeWarrior, 71
prefix files
Xcode, 83, 84
PRINT message, 122
problems, 26
procedures
CallFunction, 285
calling from an XOP, 285
CheckFunctionForm, 285, 469
DoWindowRecreationDialog, 466
GetFunctionInfo, 285, 467, 471
GetFunctionInfoFromFuncRef, 285, 468
GetIgorProcedure, 465
GetIgorProcedureList, 464
routines, 464–72
SetIgorProcedure, 466
programming utilities, 491–96
project files, 15
CodeWarrior, 68
Visual C++ 6, 90
Visual C++ 7, 95
project settings
CodeWarrior, 69
Visual C++ 6, 92
Visual C++ 7, 97
projects
creating in CodeWarrior, 67, 75
creating in guided tour, 35
creating in Visual C++ 6, 90
creating in Visual C++ 7, 95
creating in Xcode, 80, 83
debug and release configurations, 91, 96
PtrAndHand, 501
PtrToHand, 501
PutCmdLine, 485
PutCStringInHandle, 217, 220, 491


# Q


QDPointer, 527
quotes
CatPossiblyQuoteName, 460
PossiblyQuoteName, 460


# R


recompiling XOPs, 25
rectangles, 496
recursion, 137–38, 254
DoUpdate side-effect, 478
XOPCommand side-effect, 476
XOPCommand2 side-effect, 476
XOPSilentCommand side-effect, 477
RegisterClass, 251
RegisterOperation, 147, 152, 329
example, 105
release configuration
Visual C++ 6, 91

Visual C++ 7, 96
ReleaseMenu, 245, 527
relocatable block of memory, 263, 264
RenameDataFolder, 383
RenameDataFolderObject, 391
REPLACE message, 121
REPLACEALLCMDS, 485
REPLACEALLCMDSCRHIT, 485
REPLACEFIRSTCMD, 485
ResEdit, 109
RESIDENT bit, 135
SetXOPType, 326
resident XOPs
definition of, 21
external functions, 201
XOPType field in IORec, 135
Resorcerer, 109
resource files, 15
restrictions on, 288
Visual C++ 6, 91
Visual C++ 7, 96
resource includes, 109
resource XOPSupport routines, 447
resource.h file
creation of, 110
dialog resource IDs, 269
menu resource IDs, 235, 237
ResourceMenuIDToMenuHandle, 240, 394
resources, 108–11
accessing in Carbon, 536
CNTL, 275
compatibility, 518
creating, 109
creating on Windows, 109
DITL resources, 269
dlgx, 270
DLOG resources, 269
for dialogs, 269
for menus, 234–38
GetIndString, 537
GetResource, 288, 536
GetXOPIndString, 447, 537
GetXOPNamedResource, 447
GetXOPResource, 447
guided tour, 53
hdlg resources, 295
hmnu resources, 295
HMNU resources, 297
in CFM XOPs, 109
in Mach-O XOPs, 109
initialization, 20
MENU, 275
MENUHELP resources, 297
popup menus, 275
resource.h file, 110, 235, 237, 269
restrictions on, 288
SimpleGaussFit, 53
status line help, 297
STR# 1100 resource, 108, 128, 130
STR# 1101 resource, 108, 236
STR# 1160 resource, 108
STR# for balloon help, 296
UseResFile, 288, 536
version resources, 278
XMI1 1100, 234, 236
XMI1 1100 resource, 108, 296, 297
XMN1 1100, 234, 235
XMN1 1100 resource, 108
XOPC 1100 resource, 108, 115, 148
XOPCloseResFile, 537
XOPF 1100 resource, 108, 115, 182, 183
XOPI 1100 resource, 108, 110
XOPOpenResFile, 537
XOPRefNum, 447
XOP-specific, 108
XOPTypes.r, 109
XOPUseResFile, 537
XPRF resources, 489, 490
XSM1 1100, 234, 237
XSM1 1100 resource, 108
ResourceToActualItem, 241, 393
ResourceToActualMenuID, 240, 393
responding to menu selections, 239–41
result types
in external functions, 188
results
external functions, 187
string in external functions, 190, 191
ResumeUpdate, 478
RETRIEVE message, 120
REVERT_WINDOW message, 123
Rez
compiling .r files, 109
RGBColor, 462
root data folder, 379
routine descriptors
XOPDialog, 399
ROWS, 216, 336, 360
RS232
VDT2 sample XOP, 19
runtime library
Visual C++ 6, 92
Visual C++ 7, 97
runtime parameter structure, 159

# S

S_ variables
    in external operations, 174
S_fileName, 428
    SetFileLoaderOperationOutputVariables, 428
S_path, 428
    SetFileLoaderOperationOutputVariables, 428
S_waveNames, 428
    SetFileLoaderOperationOutputVariables, 428
sample XOPs, 15, 17–19
SanitizeWaveName, 456
SAVE message, 124, 132
SAVE_TYPE_xxx, 125
SAVE_WINDOW message, 123
SAVE_WINDOW_AS message, 123
SAVE_WINDOW_COPY message, 123
SAVE_WINDOW_MACRO message, 123
SAVEFILE message, 122
SAVESETTINGS message, 125, 133
SaveXOPPrefsHandle, 132, 489
saving settings, 133
ScaleClipAndRound, 543
ScaleClipAndRoundData, 453
ScaleData, 453
SELECT_ALL message, 122
SelEditItem, 403, 525
SelMacEditItem, 403
SendWinMessageToIgor, 253, 475
SendXOPA4ToIgor, 527
serial port
    VDT2 sample XOP, 19
SET_TARGET_WINDOW_NAME message, 123
SET_TARGET_WINDOW_TITLE message, 123
SetCheckBox, 400
SetCurrentDataFolder, 379
SetDataFolderObject, 387
SetDataFolderProperties, 377
SetDDouble, 403
SetDialogBalloonHelpID, 399, 526
SetDialogPort, 398, 525
SetDInt, 402
SetDItemProc, 529
SetDLong, 402
SetDText, 401
SetFileLoaderOperationOutputVariables, 428
SetFileLoaderOutputVariables, 174, 428
SETGROW message, 117
SetHandleSize, 499
    use with strings, 321
SetIgorComplexVar, 372
SetIgorFloatingVar, 372
    example, 223
SetIgorIntVar, 371
SetIgorMenuItem, 243, 392
SetIgorProcedure, 466
SetIgorStringVar, 373
    example, 223
setmenuitemtext, 503, 530
SetNaN32, 493
SetNaN64, 493
SetNVAR, 374
    example, 283
SetOperationFileLoaderOutputVariables, 174
SetOperationNumVar, 330
SetOperationStrVar, 330
SetOperationWaveRef, 167, 333, 542
SetPopItem, 407
SetPopMatch, 407
SetPopMenu, 527
SetPtrSize, 498
SetRadBut, 400
SetStandardFilePath, 528
SetSVAR, 375
    example, 283
SetTextWaveData, 218, 367, 542
settings
    byte reordering, 134
    GetPrefsState, 490
    GetXOPPrefsHandle, 490
    loading, 133
    messages for, 124–26
    SaveXOPPrefsHandle, 489
    saving, 133
    saving and loading, 133
    structure alignment, 133
    structures, 134
SetWaveDimensionLabels, 354, 542
SetWaveLock, 340
SetWaveNote, 343
SetWaveScaling, 341
    history of, 219
SetWavesStates, 344
SetWaveUnits, 342
    history of, 219
SetXOPEntry, 24, 326
    example, 105
    XOPEntry field in IORecHandle, 136
SetXOPItem, 530
SetXOPMessage, 327
SetXOPRefCon, 328
SetXOPResult, 127, 327
    example, 105
    result field in IORecHandle, 136
    returning custom errors, 130

SAVESETTINGS message, 133
SetXOPType, 326
IDLE message, 112
in external functions, 201
XOPType field in IORec, 135
SetXOPWindowIgorPositionAndState, 433
SetXOPWindowPositionAndState, 431
SetXOPWindowTitle, 431
shared library
CodeWarrior, 68
shortcuts, 20
SHOW_MENU_AT_LAUNCH, 235
SHOW_MENU_WHEN_ACTIVE, 235
ShowAndActivateXOPWindow, 430
ShowDialogWindow, 398
ShutdownDialogSettings, 272
SIGNED_INT, 450
SimpleFit sample XOP, 18, 29
SimpleGaussFit
compiling, 58
resources, 53
testing, 60
SimpleGaussFit sample XOP, 29
SimpleLoadWave sample XOP, 18
solution files, 95
SoundInput XOP, 448
SpinCursor, 434
SpinProcess, 484
recursion, 137
SS_MASK, 489
stack
overwriting, 305
starter code, 152
bug in Igor Pro 5.00 and 5.01, 541
details, 175
updating, 176
status line
XOPSetContextualHelpMessage, 488
status line help
for menus, 297–99
StdGetFile, 528
StdPutFile, 528
StoreNumericDataUsingVarName, 165, 332
StoreNumericValue, 362
StoreNumVar, 369
StoreStringDataUsingVarName, 165, 333
StoreStrVar, 370
STR# 1100 resource, 108, 128, 130
STR# 1101 resource, 108, 236, 530
XOP help file name, 293
STR# 1160 resource, 108
STR# resource
status line help, 297
STR# resources for balloon help, 296
STR_OBJECT, 376–91
CheckName, 457
CreateValidDataObjectName, 458
strchr2, 492
String
in structure parameters, 282
string expressions
in operations, 146
string parameters
external operations, 163
string variables
accessing, 368–75
content stored in handles, 220
FetchStrHandle, 370
FetchStrVar, 370
GetSVAR, 375
SetFileLoaderOperationOutputVariables, 428
SetIgorStringVar, 373
SetOperationStrVar, 330
SetSVAR, 375
StoreStringDataUsingVarName, 333
StoreStrVar, 370
StringList, 482
Variable, 368
VarNameToDataType, 331
StringList, 482
strings
disposing, 163
GetCStringFromHandle, 217, 220
GetHandleSize, 321
in handles, 321
in Igor, 193, 220
in structure parameters, 283
IsStringExpression, 335
pitfalls, 321
PutCStringInHandle, 217, 220
SetHandleSize, 321
strrchr2, 492
structure alignment, 160
changing for XOP Toolkit 5, 515
in CodeWarrior, 280
in external functions, 187
settings structures, 133
structure parameters, 281
CheckFunctionForm, 469
example in external function, 196
example in external operation, 169
extended, 172
extended example in external operation, 173
fields, 282
in external functions, 194
in external operations, 168

NVARs, 283
strings, 283
SVARs, 283
structures
expandable, 132, 134
GetNVAR, 374
GetSVAR, 375
in XOPF resource, 184
SetNVAR, 374
SetSVAR, 375
versioning, 171, 198
Xcode, 88
subclassing, 254
submenus
XSM1 1100 resource, 237
SVAR, 165
SVARs
GetSVAR, 375
in structure parameters, 282, 283
SetSVAR, 375
symbolic debugging, 318
symbolic paths
FileLoaderGetOperationFlags2, 335
FillPathMenu, 396
FillPathPopMenu, 409
GetFullPathFromSymbolicPathAndFilePath, 426
GetPathInfo2, 481
PathList, 480


# T


target settings
CodeWarrior, 69
target windows
adding, 257
TARGET_RT_MAC_MACHO, 43
CodeWarrior, 77
Xcode, 84
Task Manager
leaks, 310
technical support, 26
email, 26
FTP, 26
mailing list, 26
phone number, 26
template, 152
templates, 155
access waves, 214
for operations and functions, 292
length limit, 175
mnemonic names, 158
temporary storage access method, 213
text utility routines, 435–46
TUDemo sample XOP, 19
text waves
accessing text data, 217–18, 542
example, 140
faster access, 218
GetTextWaveData, 365
MDGetTextWavePointValue, 363
MDSetTextWavePointValue, 364
SetTextWaveData, 367
text wave type, 211
text windows
TUActivate, 437
TUClear, 439
TUClick, 439
TUCopy, 438
TUCut, 438
TUDelete, 444
TUDemo sample XOP, 19
TUDisplaySelection, 436
TUDispose, 436
TUDrawWindow, 442
TUFetchParagraphText, 444
TUFetchSelectedText, 445
TUFind, 440
TUFixEditMenu, 441
TUFixFileMenu, 441
TUGetDocInfo, 442
TUGetSelLocs, 443
TUGrow, 436
TUIdle, 437
TUIndentLeft, 440
TUIndentRight, 440
TUInsert, 443
TUInsertFile, 444
TUKey, 440
TULines, 442
TUMoveToFullSizePosition, 438
TUMoveToPreferredPosition, 437
TUNew, 435
TUNew2, 435
TUNull, 438
TUPageSetupDialog, 439
TUPaste, 439
TUPrint, 439
TUReplace, 440
TURetrieveWindow, 438
TUSelectAll, 440
TUSetSelLocs, 443
TUSetStatusArea, 446
TUSFInsertFile, 441
TUSFWriteFile, 441
TUUndo, 439

TUUpdate, 437
TUWriteFile, 444
TEXT_WAVE_TYPE, 337
type for waves, 211
THINK C, 4
threading
Visual C++ 6, 92
Visual C++ 7, 97
TickCount, 504
To Clip button, 277
To Cmd button, 277
TOCLIPID, 404
TOCMDID, 404
ToggleCheckBox, 400
TransformWindowCoordinates, 432
TRANSIENT bit, 135
SetXOPType, 326
transient XOPs
definition of, 21
new operator, 101
with windows, 252
XOPType field in IORec, 135
troubleshooting, 26
try
in C++, 100
TU windows, 254
subclassing, 254
TUDispose, 254
TUNew, 254
TUNew2, 254
window procedure, 254
TUActivate, 437
TUClear, 439
TUClick, 439
TUCopy, 438
TUCut, 438
TUDelete, 444
TUDemo sample XOP, 19
TUDisplaySelection, 436
TUDispose, 254, 436
TUDrawWindow, 442
TUFetchParagraphText, 444
TUFetchSelectedText, 445
TUFind, 440
TUFixEditMenu, 441
TUFixFileMenu, 441
TUGetDocInfo, 442
TUGetSelLocs, 443
TUGrow, 436
TUIdle, 437
TUIndentLeft, 440
TUIndentRight, 440
TUInsert, 443
TUInsertFile, 444
TUKey, 440
TULines, 442
TULoc structure, 443
TUMoveToFullSizePosition, 438
TUMoveToPreferredPosition, 437
TUNew, 254, 435
TUNew2, 254, 435
TUNull, 438
TUPageSetupDialog, 439
TUPaste, 439
TUPrint, 439
TUReplace, 440
TURetrieveWindow, 438
TUSelectAll, 440
TUSetSelLocs, 443
TUSetStatusArea, 446
TUSFInsertFile, 441
TUSFWriteFile, 441
TUUndo, 439
TUUpdate, 437
TUWriteFile, 444
types of XOPs, 21

# U

undo
menu item, 392
TUUndo, 439
UNDO message, 121
uninitialized variables, 304
UniqueName, 455
UniqueName2, 455
units, 342
for waves, 218
MDGetWaveUnits, 351
MDSetWaveUnits, 352
Unix paths
Xcode, 88
unlocking handles, 316
UNSIGNED_INT, 450
update
DoUpdate, 478
XOPCommand side-effect, 476
XOPCommand2 side-effect, 476
XOPSilentCommand side-effect, 477
UPDATE
recursion, 138
UPDATE message, 116, 250
not sent on Windows, 253
recursion, 137
updates for XOP Toolkit, 8, 65

user functions
versus external functions, 181
user-defined functions
calling from an XOP, 285
UseResFile, 288, 536
UserItem controls, 533, 534
utilities, 491–96
utility routines, 16
CmpStr, 492
DateToIgorDateInSeconds, 494
GetCStringFromHandle, 491
IgorDateInSecondsToDate, 494
IgorVersion, 486
IsINF32, 493
IsINF64, 493
IsMacOSX, 488
IsNaN32, 493
IsNaN64, 493
MacRectToWinRect, 496
MemClear, 491
MoveLockHandle, 495
PutCStringInHandle, 491
SetNaN32, 493
SetNaN64, 493
strchr2, 492
strrchr2, 492
WinInfo, 489
WinRectToMacRect, 496
XOPBeep, 495
XOPOKAlert, 495
XOPOKCancelAlert, 495
XOPYesNoAlert, 495
XOPYesNoCancelAlert, 496

# V

V_ variables
in external operations, 174
V_flag, 428
V_Flag
SetFileLoaderOperationOutputVariables, 428
VAR_GLOBAL, 368
VAR_OBJECT, 376–91
CheckName, 457
CreateValidDataObjectName, 458
Variable, 368
in structure parameters, 282
VariableList, 481
variables
accessing, 368–75
accessing data in, 220–23
commonly used XOPSupport routines, 221
data folders, 368
example making, 223
FetchNumVar, 369
FetchStrHandle, 370
FetchStrVar, 370
GetNumVarName, 335
GetNVAR, 374
GetStrVarName, 335
GetSVAR, 375
local versus global, 368
making global, 223
precision of, 368
SetFileLoaderOperationOutputVariables, 428
SetIgorComplexVar, 372
SetIgorFloatingVar, 372
SetIgorIntVar, 371
SetIgorStringVar, 373
SetNVAR, 374
SetOperationNumVar, 330
SetOperationStrVar, 330
SetSVAR, 375
setting in external operations, 330, 331, 332, 333, 428
StoreNumericDataUsingVarName, 332
StoreNumVar, 369
StoreStringDataUsingVarName, 333
StoreStrVar, 370
StringList, 482
uninitialized, 304
VAR_GLOBAL, 368
Variable, 368
VariableList, 481
VarNameToDataType, 331
VarName parameters
external operations, 165
VarNameToDataType, 331
VDT2 sample XOP, 19
version
of Igor, 140
version resources, 278
versions, 140–42
compatibility, 6
IgorVersion, 486
of Igor, 4, 5, 6, 26, 147
of structures, 171, 198
XOP Toolkit 5 Upgrade, 507
Visual C++
C++, 102
first XOP support, 5
resources, 109
structure alignment, 280
supported versions, 6, 8, 65
worksheet files, 15
XOPSupport project files, 13

Visual C++ .NET, 94–98
Visual C++ 6, 90–93
.dsp files, 90
.dsw files, 90
.rc files, 91
building SimpleFit, 32
configurations, 91
creating a new project, 90
creating project in guided tour, 45
debug configuration, 91
debugging, 93
project files, 90
project settings, 92
release configuration, 91
resource files, 91
runtime library, 92
testing installation, 12
threading, 92
workspace files, 90
Visual C++ 7, 94–98
.NET, 94
.rc files, 96
.sln files, 95
.vcproj files, 95
building SimpleFit, 33
configurations, 96
creating a new project, 95
creating project in guided tour, 49
debug configuration, 96
debugging, 98
LNK4204, 98
project files, 95
project settings, 97
release configuration, 96
resource files, 96
runtime library, 97
solution files, 95
testing installation, 12
threading, 97
warnings, 98
volume reference numbers, 267

W

warnings
Visual C++ 7, 98
WatchCursor, 434
wave assignment statements, 186
wave parameters
external operations, 164
WAVE references, 166
in external operations, 333
in structure parameters, 282
SetOperationWaveRef, 167, 542
WAVE_OBJECT, 376–91
CheckName, 457
CreateValidDataObjectName, 458
WAVE_OBJECT object type, 114
WAVE_TYPE, 188
curve fitting, 189
in XOPF resource, 184
WaveAccess sample XOP, 19, 215
WaveData, 343
C++, 99
example, 210
WaveHandleModified, 345
WaveHandlesModified, 345
WaveList, 479
WaveLock, 340
WaveModCount, 340
WaveModDate, 339
WaveModified, 345
WaveModState, 340
WaveName, 343
WaveNote, 342
WavePoints, 339
WaveRange parameters
external operations, 164
waves
accessing, 336–65
accessing data in, 207–19
accessing numeric data, 212–16
accessing text data, 217–18, 542
accessing with C++ templates, 214
CalcWaveRange, 335
ChangeWave, 338
commonly used XOPSupport routines, 207
data folders, 381
data scaling, 218
dimension scaling, 218
direct access method, 214
example making, 210
FetchNumericValue, 362
FetchWave, 338
FetchWaveFromDataFolder, 339
FileLoaderMakeWave, 427
FillWaveMenu, 395
FillWavePopMenu, 408
GetTextWaveData, 365
GetWave, 335
GetWaveDimensionLabels, 353
GetWaveList, 335
GetWaveName, 335
GetWaveRange, 335
GetWavesInfo, 344

KillWave, 338
lock state, 340
MakeWave, 337
MDAccessNumericWaveData, 355
MDChangeWave, 348
MDChangeWave2, 349, 543
MDGetDimensionLabel, 352
MDGetDPDataFromNumericWave, 359
MDGetNumericWavePointValue, 357
MDGetTextWavePointValue, 363
MDGetWaveDimensions, 347
MDGetWaveScaling, 349
MDGetWaveUnits, 351
MDMakeWave, 346
MDSetDimensionLabel, 353
MDSetNumericWavePointValue, 358
MDSetTextWavePointValue, 364
MDSetWaveScaling, 350
MDSetWaveUnits, 352
MDStoreDPDataInNumericWave, 360
number types, 211
OBJINUSE message, 114
organization of numeric data, 216
point access method, 212
ranges of, 164
SanitizeWaveName, 456
SetFileLoaderOutputVariables, 428
SetOperationWaveRef, 333, 542
SetTextWaveData, 367
SetWaveDimensionLabels, 354
SetWaveLock, 340
SetWaveNote, 343
SetWaveScaling, 341
SetWavesStates, 344
SetWaveUnits, 342
speed of accessing numeric data, 215
StoreNumericValue, 362
temporary storage access method, 213
text wave example, 140
text wave type, 211
units, 218
WaveAccess example, 19
WaveData, 343
WaveHandleModified, 345
WaveHandlesModified, 345
WaveList, 479
WaveLock, 340
WaveModCount, 340
WaveModDate, 339
WaveModified, 345
WaveModState, 340
WaveName, 343
WaveNote, 342
WavePoints, 339
WaveScaling, 341
WaveType, 339
WaveUnits, 342
XOPWaveAccess.c, 13
WaveScaling, 341
history of, 219
WaveType, 339
WaveUnits, 342
history of, 219
wctb resources, 518
WIND resources, 518
window procedures, 252
for TU windows, 254
WINDOW_MOVED message, 118
not sent on Windows, 253
WindowPtr, 116
XOP_WINDOW_REFs, 249
windows
activating, 116
adding, 249–58
adding on Macintosh, 250
adding on Windows, 251
ArrowCursor, 434
captions, 431
clearing, 121
click events, 118
close type code, 117
closing, 117
content region, 255, 256
coordinates, 255
copying, 121
CreateWindowEx, 251
CreateXOPWindow, 252
CreateXOPWindowClass, 252
creating on Macintosh, 250
cursors, 118
cutting, 120
destroying an MDI window, 252
DestroyXOPWindow, 252
disposing on Macintosh, 250
DUPLICATE message, 122
EXPORT_GRAPHICS message, 122
FillWindowPopMenu, 409
FillWinMenu, 396
finding, 121
GET_TARGET_WINDOW_NAME message, 123
GET_TARGET_WINDOW_REF message, 123
GetActiveWindowRef, 430
getting Igor client HWND, 473
GetXOPWindow, 430
GetXOPWindowIgorPositionAndState, 433
GetXOPWindowPositionAndState, 431

HandCursor, 434
help, 300
HideAndDeactivateXOPWindow, 431
IBeamCursor, 434
Igor window coordinates, 255
indenting, 121
inserting files in, 122
IsXOPWindowActive, 430
key events, 118
MDI child window menu bar, 253
MDI child windows, 251, 252
MDI window coordinates, 432
MDI window position, 255, 431
messages, 24
messages for, 116–23
move to full position message, 119
move to preferred position message, 119
moved message, 118
null events, 118
page setup, 122
pasting, 121
printing, 122
recursion problems, 254
RegisterClass, 251
replacing, 121
resizing, 116, 117, 436
retrieve message, 120
REVERT_WINDOW message, 123
SAVE_WINDOW message, 123
SAVE_WINDOW_MACRO message, 123
saving files from, 122
SELECT_ALL message, 122
SendWinMessageToIgor, 253, 475
SET_TARGET_WINDOW_NAME message, 123
SET_TARGET_WINDOW_TITLE message, 123
SetXOPWindowIgorPositionAndState, 433
SetXOPWindowPositionAndState, 431
SetXOPWindowTitle, 431
ShowAndActivateXOPWindow, 430
SpinCursor, 434
support routines, 430–34
target windows, 257
text, 435–46
TransformWindowCoordinates, 432
transient XOPs, 252
TU windows, 254
TUNew, 254
TUNew2, 254
undoing, 121
updating, 116
WatchCursor, 434
window procedures, 252
windowKind field, 250, 430
WindowXOP1 sample XOP, 17
WinList, 480
XOP_WINDOW_REF, 116
XOPSetContextualHelpMessage, 488
XOPWindowProc, 252
zooming, 116, 436

Windows OS
- IgorClientHWND, 473
- IgorModule, 473
- SendWinMessageToIgor, 475
- WindowsErrorToIgorError, 474
- WMGetLastError, 473
- XOPModule, 473

Windows OS error codes, 128–30

Windows platform
- compatibility with Igor Pro 5, 5
- first Igor release, 5
- requires Igor Pro 4, 6
- support routines, 473–75
- supported OS versions, 6
- XOP Toolkit 5 requirements, 6

WindowsErrorToIgorError, 128–30, 474
WindowXOP1 sample XOP, 17
WindowXOP1 XOP
- balloon help example, 296

WinHelp, 300
WinInfo, 489
WinList, 480
WinRectToMacRect, 496
WinToMacPath, 268, 412
WM_ACTIVATE message, 251
WM_CHAR message, 251, 253
- in TU windows, 254

WM_COMMAND message, 239, 251, 253
WM_CONTEXTMENU, 300
WM_DESTROY message, 252
WM_INITMENU message, 242, 251
WM_KEY message, 251, 253
WM_LBUTTONDOWN message, 253
WM_MDIACTIVATE message, 253
- in TU windows, 254

WM_MDIDESTROY message, 252
WM_MOVE message, 253
WM_PAINT message, 251, 253
- in TU windows, 254
- recursion, 137

WM_RBUTTONDOWN message, 253
WM_SIZE message, 253
WMDeleteMenuItems, 394
WMGetLastError, 128–30, 473
working directory refNums, 267
worksheet files, 15
workspace files, 90

World-Wide Web, 26

X

X scaling, 341
  MDGetWaveScaling, 349
  MDSetWaveScaling, 350
x86, 5
Xcode, 79–89
  .exp files, 81, 84
  balloon help, 88
  building SimpleFit, 31
  bundles, 82
  C++, 84, 102
  command help, 34
  creating a new project, 80, 83
  creating project in guided tour, 41
  debugging, 86
  DEV_SYS_CODE, 84
  Exports.exp file, 43
  fopen function, 88
  frameworks, 83
  help files, 88
  icon, 85
  IGR0 XOP file creator, 84
  info.plist, 83
  IXOP XOP file type, 44, 84
  LaunchCFMApp, 83, 86
  Mach-O, 80, 83
  main function, 84, 102
  packages, 81, 85
  PkgInfo file, 85
  POSIX paths, 88
  prefix files, 83, 84
  structures, 88
  supported versions, 8, 65
  TARGET_RT_MAC_MACHO, 43, 84
  testing installation, 11
  Unix paths, 88
  versus CodeWarrior, 6
  XOPSupport project files, 13
XFUNC1 sample XOP, 17
XFUNC1 XOP
  description, 182
  direct functions, 202
  XOPF 1100 resource example, 183
XFUNC1Add
  example external function, 182
XFUNC1ComplexConjugate
  example external function, 182, 189
XFUNC1Div
  example external function, 182
  invoking, 186
  parameter types, 188
  parameters and result, 187
XFUNC2 sample XOP, 17
XFUNC2 XOP
  description, 182
  direct functions, 202
  XOPF 1100 resource, 188
XFUNC3 sample XOP, 17
XFUNC3 XOP
  description, 182
  XOPF 1100 resource, 190
XFUNCs. (see external functions)
XMI1 1100 resource, 108, 234, 236
  adding balloon help, 296
  itemFlags field, 242
  status line help, 297
XMN1 1100 resource, 108, 234, 235
XOP errors, 128
XOP files, 20
XOP protocol, 20–21
  compatibility, 140
  GetXOPItem, 328
  GetXOPMessage, 327
  GetXOPRefCon, 328
  GetXOPResult, 327
  GetXOPStatus, 328
  SetXOPEntry, 326
  SetXOPMessage, 327
  SetXOPRefCon, 328
  SetXOPResult, 327
  SetXOPType, 326
  version, 111
  XOPInit, 326
XOP protocol version, 141
XOP resources, 108–11
XOP Toolkit
  Carbon, 521
  installing, 10
  new features, 509
  new XOPSupport routines, 516
  overview, 13–16
  Release 5.00, 541
  Release 5.03, 541
  Release 5.04, 542
  sample XOPs, 15
  updates, 8, 65
  version, 111
  version 5, 5
  version 5 upgrade, 507
XOP Toolkit number type codes, 450, 452, 453
XOP.h, 13
XOP_DIALOG_REF

dialogs, 270
XOP_GLOBALS_ARE_A4_BASED, 524
XOP_SET_STRUCT_PACKING, 515
XOP_TOOLKIT_VERSION, 111
XOP_VERSION, 111, 141
XOP_WINDOW_REF, 116
XOP_WINDOW_REFs, 249
XOP1 sample XOP, 17
XOPAtEndOfFile, 420
XOPBeep, 495
XOPC 1100 resource, 105, 108, 115, 148
XOPCloseFile, 418
XOPCloseResFile, 528, 537
XOPCommand, 476
recursion, 137, 138
XOPCommand2, 476
XOPCreateFile, 417
XOPDeleteFile, 417
XOPDialog, 399, 529
XOPDisplayHelpTopic, 300, 486
XOPEntry, 20, 24, 107, 112, 136
FUNCTION message, 201
messages, 21, 22
XOPF 1100 resource, 108, 115, 201
complex parameters, 189
defined, 183
inspected by Igor Pro, 182
string parameters, 190
XFUNC2 XOP example, 188
XOPGetFilePosition, 420
XOPI
changing for XOP Toolkit 5, 508
XOPI 1100 resource, 105, 108, 110
XOP protocol version, 141
XOPInit, 24, 326
example, 105
XOPRecHandle global, 135
XOPINITING status bit
MENUITEM message, 241
XOPModule, 473
XOPNotice, 478
for debugging, 319
pitfalls, 320
XOPNumberOfBytesInFile, 420
XOPOKAlert, 131, 495
XOPOKCancelAlert, 131, 495
XOPOp operation category, 149
XOPOpenFile, 417
XOPOpenFileDialog, 269, 421, 526, 542
XOPOpenResFile, 528, 537
XOPReadFile, 418
XOPReadFile2, 418
XOPReadLine, 419
XOPRecHandle global, 135
XOPRefNum, 447
XOPResNotice, 479
XOPs. (see external operations)
name conventions, 7
XOPS 1100 resource, 133
XOPSaveFileDialog, 269, 423, 526, 542
XOPSetContextualHelpMessage, 488, 530
XOPSetFilePosition, 420
XOPSilentCommand, 477
recursion, 137, 138
XOPStandardHeaders.h, 13
XOPStructureAlignmentTwoByte.h, 160
XOPSupport
callbacks, 16, 21
LNK4204 warnings, 98
new routines, 516
utility routines, 16
XOPSupport folder, 16
XOPSupport folder, 13, 16
XOPSupport project files, 13
XOPSupport.c, 13
XOPSupport.h, 13
XOPTypes.r, 109
XOPUseResFile, 528, 537
XOPWaveAccess.c, 13
XOPWindowProc, 252
XOPWriteFile, 419
XOPYesNoAlert, 131, 495
XOPYesNoCancelAlert, 131, 496
XPRF resources, 489, 490
XSM1 1100 resource, 108, 234, 237
xstrcat
example external function, 182, 191
XOPF 1100 resource, 190
xstrcat0 external function, 17
xstrcat1 external function, 17