

IGOR XOP TOOLKIT

REFERENCE MANUAL

Version 6.40

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.

Notice

All brand and product names are trademarks or registered trademarks of their respective companies.

Manual Revision: October 27, 2015 (6.40)

© Copyright 2015 WaveMetrics, Inc. All rights reserved.

WaveMetrics, Inc.
PO Box 2088
Lake Oswego, OR 97035
USA

Voice: 503-620-3001
FAX: 503-620-6754
Email: support@wavemetrics.com, sales@wavemetrics.com
Web: <http://www.wavemetrics.com/>

Table of Contents

Chapter 1 - Introduction to XOPs

About This Manual	1
What is an XOP?.....	2
Who Can Write an XOP?.....	2
A Brief History of Igor	2
XOP Toolkit 6.4.....	4
Macintosh and Windows XOPs	4
Development Systems.....	4
Activating XOPs Using the Igor Extensions Folder	5
Installing the XOP Toolkit.....	5
XOP Toolkit Overview	6
XOPSupport.....	9
The Sample XOPs.....	9
How Igor Integrates XOPs.....	11
The Basic Structure of an XOP.....	13
Preparing to Write an XOP	15
Technical Support	15

Chapter 2 - Guided Tour

Overview.....	17
Activating XOPs	17
What We Will Do	18
Installation	18
Building SimpleFit.....	18
Testing SimpleFit.....	20
Creating a New Project	21
Building the New Project in Xcode 6	23
Changing the Help	25
Changing and Compiling the Code.....	26
Testing SimpleGaussFit.....	27
Touring the Code	29
Where To Go From Here	30

Chapter 3 - Development Systems

Overview.....	31
XOPs in Xcode	31
XOP Projects in Xcode	32
Xcode 3 Project Settings.....	33
Xcode 4 Project Settings.....	33

Xcode 6 Project Settings	34
Debugging an XOP in Xcode 6	40
Converting an Old Xcode Project	41
XOPs in Visual C++ 2008 through 2013	43
XOPs in Visual C++ 2015	43
XOP Projects in Visual C++ 2015	43
Visual C++ 2015 Project Settings	44
Visual C++ 2015 XOPSupport Warnings	46
Debugging a Visual C++ 2015 XOP	46
Converting an Old Visual C++ Project	47

Chapter 4 - Igor/XOP Interactions

Overview	49
XOP Resources	51
How Igor Calls an XOP	53
Basic XOP Messages	54
Messages for XOPs that Save and Load Settings	56
XOP Errors	57
XOPs and Preferences	61
XOPs and Experiments	61
The IORecHandle	62
Messages, Arguments and Results	63
Handling Recursion	64
Data Sharing	65
Igor/XOP Compatibility Issues	66

Chapter 5 - Adding Operations

Overview	69
The XOPC 1100 Resource	70
Choose a Distinctive Operation Name	72
Operation Handler	72
Creating Starter Code	72
Operation Parameters	73
The Command Template	74
Optional Parameters	75
Mnemonic Names	77
The Runtime Parameter Structure	77
Qualified Keywords	80
String Parameters	81
Name Parameters	81
Double Name Parameters	82

Wave Parameters	82
Wave Range Parameters	82
VarName Parameters	83
Data Folder Reference Parameters.....	84
DataFolderAndName Parameters	84
Structure Parameters	87
Runtime Output Variables	92
Starter Code Details	92
Updating Starter Code	93
Thread-Safe External Operations.....	93
External Operations and Macintosh Resources	95
Operation Handler Checklist	95

Chapter 6 - Adding Functions

Overview.....	97
External Function Examples	97
Adding an External Function to Igor	98
The XOPF 1100 Resource	98
Choose a Distinctive Function Name	100
Invoking an External Function	100
External Function Parameters and Results	101
Parameter and Result Types	102
Wave Parameters	103
Wave Results	103
Data Folder Reference Parameters.....	104
Data Folder Reference Results	104
Data Folder Reference Example	105
Complex Parameters and Results	106
Strings Parameters and Results	106
Structure Parameters	109
Pass-By-Reference Parameters	112
Keep External Functions in Memory	113
FUNCTION Message Versus Direct Methods	114
Thread-Safe External Functions	114
External Functions and Macintosh Resources	116
Error Checking and Reporting.....	116

Chapter 7 - Accessing Igor Data

Overview.....	119
Waves.....	119
Numeric and String Variables.....	135

Dealing With Data Folders	138
---------------------------------	-----

Chapter 8 - Adding Menus and Menu Items

Overview.....	143
XOPSupport Menu Routines	143
Adding Menus and Menu Items.....	145
Responding to Menu Selections	148
Enabling and Disabling Menu Items	150
Advanced XOP Menu Issues	152

Chapter 9 - Adding Windows

Overview.....	153
Adding a Simple Window on Macintosh.....	153
Adding a Simple Window on Windows	154
TU ("Text Utility") Windows	156
Messages for XOPs with Windows	156
Recursion Problems	162
Igor Window Coordinates.....	162
Adding XOP Target Windows.....	164
Activating, Showing and Hiding Windows	164

Chapter 10 - Multithreading

Multithreading With XOPs.....	167
Thread-safe External Operations and External Functions	167
Three Kinds of Threads	168
Two Kinds of Igor Preemptive Threads	168
Igor Messages and Threads.....	168
Writing Thread-safe Code	168
Igor Preemptive Thread Data Environment.....	169
Waves Passed to Igor Threads as Parameters.....	169
Categories of XOPSupport Routines	170
Igor-Thread-Safe Callbacks.....	170
Thread-Safety of Callbacks from an Igor Preemptive Thread.....	170
Thread-Safety With Older Versions of Igor	171
Thread-Safety With Igor Pro 6.20 or Later.....	172
You Can Not Do Callbacks From Private Threads.....	172
Determining if You are Running in a Preemptive Thread.....	173
Strategies For Threaded Data Acquisition.....	174

Chapter 11 - 64-bit XOPs

Overview.....	181
---------------	-----

Using Visual C++ For 64-bit Development.....	182
Activating a 64-bit XOP	190
Data Types For 64-bit Compatibility.....	190
Programming For 64-bit Compatibility	191

Chapter 12 - Other Programming Topics

Macintosh Memory Management.....	197
Techniques for Cross-Platform Development	201
File I/O.....	201
File Path Conversions	202
Alerts and Message Boxes.....	203
Adding Dialogs	203
Adding Version Resources	209
Structure Alignment.....	209
Using Igor Structures as Parameters.....	210
Calling User-Defined and External Functions.....	213
The XOPMain Function.....	215
Checking For Aborts.....	216
Writing XOPs in C++	217

Chapter 13 - Providing Help

Overview.....	219
Igor Pro Help File	219
Help for External Operations and Functions	220
Macintosh Balloon Help	220
Igor Tips For Macintosh XOP Menu Items	221
Status Line Help For Windows XOP Menu Items	221
General Help For XOP Dialogs and Windows.....	223

Chapter 14 - Debugging

Overview.....	225
Programming Problems	225
Debugging Techniques	236
Avoiding Common Pitfalls	237

Chapter 15 - XOPSupport Routines

About XOPSupport Routines.....	242
Routines for Communicating with Igor.....	243
Operation Handler Routines	246
Routines for Accessing Waves	257
Routines for Accessing Variables.....	289

Routines for Accessing Data Folders.....	298
Routines for XOPs with Menu Items.....	314
Routines for XOPs that Have Dialogs	325
Routines for XOPs that Access Files	338
Routines for File-Loader XOPs	353
Routines for XOPs with Windows.....	357
Routines for XOPs with Text Windows	363
Routines for Dealing with Resources	376
Routines for XOPs That Use FIFOs	378
Numeric Conversion Routines.....	380
Routines for Dealing With Object Names	384
Color Table Routines	390
Command Window	393
Routines for Dealing With Igor Procedures	397
Multithreading Support Routines.....	406
Windows-Specific Routines	409
Miscellaneous Routines	412
Programming Utilities.....	427
Macintosh Emulation Routines.....	435

Appendix A - XOP Toolkit 6 Upgrade Notes

Overview.....	441
Development Systems Supported by XOP Toolkit 6	441
Create a Copy of Your Existing XOPs	442
Changes You Must Make	442
New Features	443
XOPStandardHeaders Changes	443
Organizational Changes.....	443
Structure Alignment in XOP Toolkit 6.....	443
Removed XOPSupport Routines	445
Renamed XOPSupport Routines	447
Changed XOPSupport Routines	447
New XOPSupport Routines	448
Miscellaneous Changes.....	450
Changing the main Function.....	451
Checking the Igor Version.....	452
Remove Handle Locking Code.....	452
Callbacks From Igor Threads	452
64-bit XOPs	453
Changes For 64-bit Compatibility	453
Sample XOP Source Files Converted To C++	454

Appendix B - Project Changes For XOP Toolkit 6

Overview	457
Xcode Project Changes For XOP Toolkit 6.....	457
Visual C++ Project Changes For XOP Toolkit 6	460
Modifying Your VC2015 Project For XOP Toolkit 6	461

Appendix C - XOP Toolkit 6 Release History

Overview	483
Release 6.00	483
Release 6.01	483
Release 6.02	483
Release 6.30	483
Release 6.40	485

Index

Chapter 1

Introduction to XOPs

About This Manual

This manual describes version 6.40 of the Igor XOP Toolkit. It focuses on development using Xcode 6 on Macintosh and Visual C++ 2015 on Windows. We removed instructions for older development systems from this version of the manual. If you must use an older development system, download the XOP Toolkit 6.30 manual from:

<ftp://ftp.wavemetrics.net/IgorXOPToolkit/Manual/Old/XOPToolkitManual630.pdf>

XOP Toolkit 6.40 creates XOPs that run with Igor Pro 6.20 or later.

If you are new to XOP programming, we recommend that you start by reading the first four chapters of this manual.

This chapter provides an overview, orientation and background information.

Chapter 2 is a Guided Tour that takes you through the process of building and running an XOP.

Chapter 3 discusses the settings and techniques used with the Xcode and Visual C++ development systems. You need to read only the section that pertains to your development system.

Chapter 4 explains the basics of how Igor and XOPs interact.

After reading the first four chapters, you can select which of the remaining chapters you need to read depending on what your XOP is required to do.

Most XOP programmers will need to read Chapter 5, which explains how to add an external operation to Igor, or Chapter 6, which explains how to add an external function.

Chapter 7 explains how to access Igor data (data folders, waves and variables).

Chapters 8 through 13 discuss various topics such as adding menus, dialogs, windows and help as well as multithreading and 64-bit XOPs.

Chapter 14 discusses debugging and how to avoid bugs using solid coding practices.

Chapter 15 lists each of the XOP Toolkit routines available for you to call to interact with Igor or to perform other tasks.

If you are upgrading an XOP that was created with XOP Toolkit 5, we recommend that you read this chapter and then **XOP Toolkit 6 Upgrade Notes** on page 441. As explained there, 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 6.

For a list of changes to XOP Toolkit 6 since release 6.00 see **XOP Toolkit 6 Release History** on page 483.

If you are updating from a previous XOP Toolkit 6 release see **Release 6.30** on page 483 and **Release 6.40** on page 485.

What is an XOP?

An XOP is a relatively small piece of C or C++ code that adds functionality to Igor Pro.

XOPs can be very simple or quite elaborate. A simple XOP might, for example, add an 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”. We sometimes call XOPs “extensions” or “Igor extensions”.

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 by cloning a sample XOP project supplied by WaveMetrics. After modifying the clone, 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 define 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 concepts such as waves, command line operations, procedures and experiments.

Writing a simple XOP that just adds an operation or function to Igor requires advanced-beginner to intermediate programming skills. If you have less experience, you will need more patience.

Writing an elaborate XOP, especially one that adds windows and/or dialogs, requires intermediate programming skills and some knowledge of graphical user interface programming.

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). Igor 1.0 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 user-defined functions. In April of 1994 WaveMetrics released XOP Toolkit 2.0 which supported XOP development under MPW and THINK C. It also supported external functions for the first time.

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 supported 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 meant that any Macintosh XOP that was to run with Igor Pro 5 had to be revised to use Apple's Carbon API. The Windows version of Igor Pro 5.0 required Windows 98, Windows ME, Windows 2000 or Windows XP. Support for Windows 95 and Windows NT was dropped.

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

Igor Pro 5 was also the first version of Igor to support Mach-O XOPs. Mach-O is the binary format for an executable on OS X. Previously Macintosh XOPs used the OS 9 CFM format.

In January of 2004, WaveMetrics released XOP Toolkit 5. It supported Macintosh development in CodeWarrior and Xcode and Windows development using Visual C++ 6, 7 (2003) and 8 (2005). The main new feature was Operation Handler support. It also supported pass-by-reference parameters with external functions.

Support for structure parameters was added in XOP Toolkit 5.04 in July, 2005.

In the spring of 2006, WaveMetrics released a beta version of Igor that ran native on Intel-based Macintosh computers. At the same time, XOP Toolkit 5.07 was released, adding support for Intel-based Macintosh XOPs.

In January of 2007, WaveMetrics released Igor Pro 6.0. No new version of the XOP Toolkit was released. Igor Pro 6.0 ran on Mac OS X 10.3.9 or later and on Windows 2000, Windows XP and Windows VISTA.

In June of 2009, WaveMetrics released Igor Pro 6.1. No new version of the XOP Toolkit was released. Igor Pro 6.1 ran on Mac OS X 10.4 or later and on Windows XP, Windows VISTA and Windows 7.

In August of 2010, WaveMetrics released Igor Pro 6.2 and XOP Toolkit 5.09. Igor Pro 6.2 ran on Mac OS X 10.4 or later and on Windows XP, Windows VISTA and Windows 7. In XOP Toolkit 5.09, documentation related to obsolete technologies (e.g., Mac OS 9, CodeWarrior and Visual C++ 6) was removed.

Also in October of 2010, WaveMetrics released XOP Toolkit 6.00. XOP Toolkit 6 supported the creation of 64-bit Windows XOPs to run with the 64-bit Windows version of Igor. It required Igor Pro 6.00 or later. When running with Igor Pro 6.20 or later, an XOP created with XOP Toolkit 6 facilitated the creation of multi-threaded external operations and functions by allowing callbacks to Igor from an external operation or function called from an Igor thread, something that was not permitted before.

Chapter 1 — Introduction to XOPs

For XOP Toolkit 6, all sample source files were converted from C to C++. The programming techniques in the sample XOPs are still C techniques. The XOPSupport library remains C.

If you are writing a new XOP, it is recommended that you use XOP Toolkit 6. It is also recommended that you port existing XOPs to XOP Toolkit 6 as the opportunity arises.

In February of 2013, WaveMetrics released Igor 6.30 which ran on Mac OS X 10.4 or later and on Windows XP or later. That same month XOP Toolkit 6.30 was released. Sample projects and documentation for Visual C++ 2012 were added. All sample XOPs were updated for compatibility with Xcode 4.6 and the LLVM compiler which Apple chose to replace the GCC compiler. This required source code changes which in turn require Igor Pro 6.20 or later.

In October of 2015, WaveMetrics released XOP Toolkit 6.40. This release added Xcode 6 and Visual C++ 2015 sample projects. It also added a new Igor7-compatible set of routines for supporting XOP menus. See **Changes To XOP Menu Support** on page 486. In XOP Toolkit 6.40, documentation related to older development systems was removed.

XOP Toolkit 6.4

XOPs created with XOP Toolkit 6.4 run with Igor Pro 6.20 or later. Some features require later versions of Igor. This is noted in the documentation where applicable. See **Release 6.30** on page 483 and **Release 6.40** on page 485 for details.

The section **Igor/XOP Compatibility Issues** on page 66 explains how you can check which version of Igor you are running with.

Macintosh and Windows XOPs

XOP Toolkit 6 supports development of 32-bit Macintosh XOPs for PowerPC and Intel processors. On Windows it supports development of 32-bit and 64-bit 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. But XOPs still need to be compiled separately for each platform.

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

Macintosh XOPs are “packages” (also called “bundles”) and consist of a folder (e.g., XFUNC1.xop) and sub-folders and files containing the executable code, resources used by the XOP and information about the XOP (the info.plist file). To see the contents of a Macintosh package, right-click it in the Finder and choose Show Package Contents.

Development Systems

This table lists the development systems supported by XOP Toolkit 6.4

Development System	XOPs Run On	Dev Sys Files In	Notes
Xcode 6.1 - 6.4	Mac OS X 10.6 or later	<project>/Xcode6	Available from the Macintosh App store or through the Macintosh Developer Program.
Mac OS 10.9 or 10.10	Intel only		
Visual C++ 2015 (VC2015)	Windows 7 or later	<project>/VC2015	Any edition is OK.

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

On Macintosh, the sample projects were successfully tested with Xcode 7.01.

The sample projects will most-likely work with newer development system versions.

XOP Toolkit 6.4 includes project files for older development systems but this manual does not discuss them.

On Windows, the free Express Edition of Visual C++ is sufficient for most XOP development.

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.

Activating XOPs Using the Igor Extensions Folder

In order for Igor to recognize and load an XOP, it must be “activated”. You activate an XOP by putting it, or an alias or shortcut pointing to it, in the “Igor Extensions” folder.

There is an Igor Extensions folder in the Igor Pro Folder but you should not use this to activate XOPs. Instead use the Igor Extensions folders in the “Igor Pro User Files” folder. This folder is created by Igor Pro 6.1 or later the first time you run the program.

This table shows the default location of the Igor Extensions folders on each operating system:

Operating System	Path
Mac OS X	/Users/<user>/Documents/WaveMetrics/Igor Pro 6 User Files/Igor Extensions
Windows	C:\Users\<user>\Documents\WaveMetrics\Igor Pro 6 User Files\Igor Extensions

You can display the Igor Pro User Files folder on the desktop by choosing Help→Show Igor Pro User Files while running Igor.

The recommended way to activate an XOP is to put the XOP, or an alias or shortcut pointing to it, in the Igor Extensions folder inside the Igor Pro User Files folder. This is recommended because changing the Program Files folder on Windows causes problems and also because it is best for purposes of backup and updating if you leave the Igor Pro Folder as originally installed.

If you are activating a WaveMetrics XOP that resides in “Igor Pro Folder/More Extensions”, make an alias (*Macintosh*) or shortcut (*Windows*) for the XOP and put the alias or shortcut in Igor Pro User Files/Igor Extensions.

If you are activating your own XOP or a third-party XOP, you can put it directly in Igor Pro User Files/Igor Extensions or you can use an alias or shortcut if you prefer.

Installing the XOP Toolkit

The XOP Toolkit comes in a folder named “XOP Toolkit 6” which you can put anywhere on your hard disk. If you have no other preference, a good place to put it is in your “Igor Pro User Files” folder. To find that folder, run Igor and choose Help→Show Igor Pro User Files.

The XOP Toolkit is delivered via electronic download only. You receive download instructions from WaveMetrics when you purchase the toolkit.

After downloading the XOP Toolkit, double-click the XOPToolkit6.dmg (*Macintosh*) or XOPToolkit6.exe (*Windows*) and install the “XOP Toolkit 6” folder on your hard disk.

All of the sample XOPs, the XOPSupport library and other support files are stored in a folder named IgorXOPs6 inside the “XOP Toolkit 6” folder.

Testing the Macintosh Installation

These instructions apply to Xcode 6 which requires Mac OS X 10.9 or later. (They have also been tested in Xcode 7.01.)

1. In the Finder, open the IgorXOPs6/XFUNC1/Xcode6 folder.
2. Open the XFUNC1.xcodeproj project in Xcode 6.
3. Choose File→Project Settings.
4. Click the Advanced button.
5. Click the Legacy radio button.
6. Click Done to close the Advanced Project Settings dialog.
7. Click Done to close the Project Settings dialog.
8. Choose Build from the Xcode 6 Product menu. This creates the XFUNC1.xop package in the IgorXOPs6/XFUNC1/Xcode6/build/Debug. Don't worry if you got some warnings from the compiler during the build.
9. Make an alias from the XFUNC1.xop package and drag the alias into your Igor Extensions folder.
10. Launch Igor Pro.
11. Execute the following command:

```
Print XFUNC1Add(3,4)
```
12. Igor should print "7" in the history area.
13. If Igor displayed an error message, make sure that you activated XFUNC1.xop correctly. See **Activating XOPs Using the Igor Extensions Folder** on page 5 for details.

Testing the Windows Installation

These instructions apply to Visual C++ 2015.

1. In the Windows desktop, open the IgorXOPs6\XFUNC1\VC2015 folder.
2. Open the XFUNC1.sln file in Visual C++ 2015.
3. Choose Build Solution from the Build menu. This creates the XFUNC1.xop file in the IgorXOPs6\XFUNC1\VC2015 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, make sure that you activated XFUNC1.xop correctly. See **Activating XOPs Using the Igor Extensions Folder** on page 5 for details.

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 IgorXOPs6 folder.

The IgorXOPs6 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 the supported development systems.
- The C files and headers used to make those libraries.
- For Windows, the IGOR.lib and IGOR64.lib files which provides Macintosh emulation and utility routines.
- 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 15.

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 and 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 menu items, numeric type codes, and other items culled from Igor's own source code. This file is included in XOPStandardHeaders.h.
IgorErrors.h	#defines for Igor error codes. This file is included in XOPStandardHeaders.h.
/Xcode6/libXOPSupport.a	The Xcode 6 XOPSupport library for Mac OS X XOPs. Contains Intel code only. Requires OS X 10.6 or later.
\VC2015\XOPSupport.lib	The Visual C++ 2015 XOPSupport library for Windows XOPs.
\VC2015\XOPSupport64.lib	
IGOR.lib	Allows a Windows XOP to call Macintosh emulation and other utility routines that are implemented in the Igor.exe file on Windows.
IGOR64.lib	

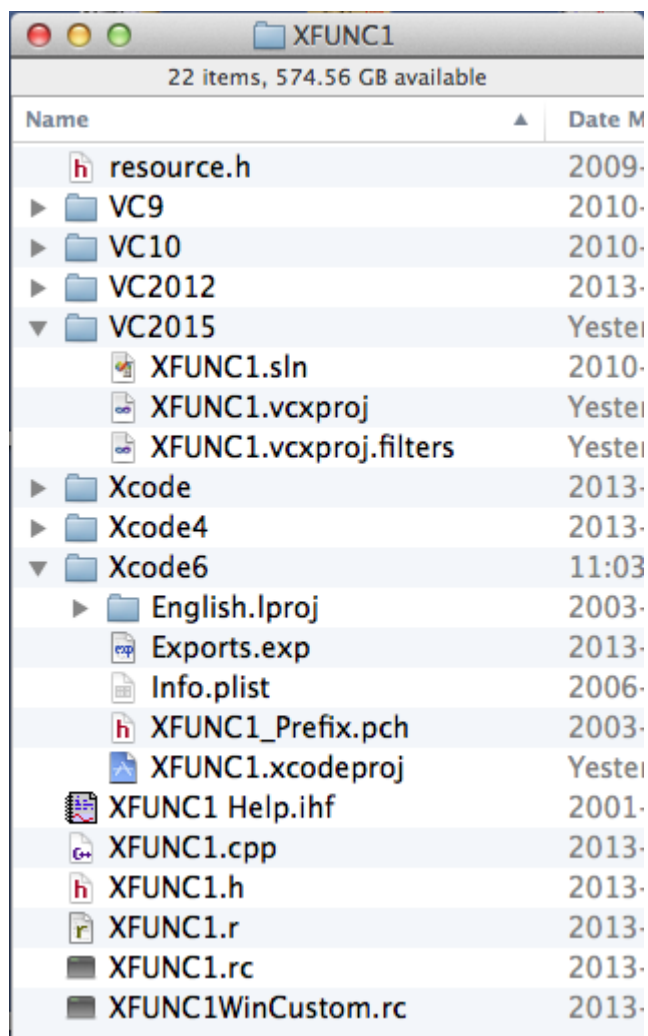
XOPSupport.lib and IGOR.lib are used when building 32-bit XOPs only.

XOPSupport64.lib and IGOR64.lib are used when building 64-bit XOPs only.

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:



Each sample XOP folder contains:

File	Example
A C++ file containing the code for the XOP.	XFUNC1.cpp
A header file containing #defines and other declarations.	XFUNC1.h
A resource file containing a text description of the XOP's resources.	XFUNC1.r (<i>Macintosh</i>) or XFUNC1.rc (<i>Windows</i>)
A "custom" resource file containing a text description of XOP-specific resources (<i>Windows</i> only)	XFUNC1WinCustom.rc
A resource header file containing defines used in the main resource file (<i>Windows</i> only)	resource.h

File	Example
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
A “project”, and/or “solution” file.	XFUNC1.xcodeproj (Xcode)
In Visual C++, open the solution file (.sln) to work with the project.	XFUNC1.sln (VC) XFUNC1.vcxproj (VC) XFUNC1.vcxproj.filters (VC)

When you compile an XOP, the compiler generates executable code from the C++ files. It then creates resources from the resource file. It writes the executable code and resources to the output executable XOP file on Windows or to the package on Macintosh. The projects are configured to store the compiled XOP in the development-system-specific folder (Xcode6 or VC2015).

XOPSupport

The XOPSupport folder contains the source code for the heart of the XOP Toolkit. These routines, described in detail in Chapter 15, are used by all XOPs. 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.

Callbacks are routines that execute code in Igor to perform services required by the XOP. There are callbacks for creating and manipulating waves and variables and for many other purposes.

Utilities are routines in the XOPSupport library itself 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 or IGOR64.lib. The callbacks that go directly to Igor have to do with Windows-specific functions (e.g., GetIgorClientHWND) 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.

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 that 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 Igor's Misc menu and a menu item with a submenu to Igor's 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.cpp 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 numeric type.

GBLoadWave is a good starting point for XOPs that import binary data to Igor.

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.

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 that 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's “Operation Handler” feature so that its operations can be called from user functions as well as from macros. Prior to the introduction of Operation Handler in 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 and aliases (*Macintosh*) or shortcuts (*Windows*) that point to XOPs.

Macintosh 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.

Chapter 1 — Introduction to XOPs

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.

The first time Igor calls the XOP, it calls the XOP's XOPMain function and passes to it 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 stored in a global variable by the XOPInit XOPSupport routine.

Among other things, the IORecHandle contains a message field which tells the XOP what it needs to do. You do not 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. (Macintosh-style handles are described under **Data Sharing** on page 65.)

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. There is a group of messages for XOPs that 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. An XOP that just adds an operation or function to Igor can ignore all messages except for INIT.

With three important exceptions, Igor always communicates with your XOP by passing a message to your XOPEntry routine. The exceptions are:

Event	How XOP Is Called
INIT message	Igor sends the INIT message by calling your XOPMain function.
Call external operation	Igor directly calls a C function that you designate (details in Chapter 5)
Call external function	Igor directly calls a C function that you designate (details in Chapter 6)

Once the XOP has received an applicable message it needs to do something. This sometimes requires calling Igor back to ask it to do something like create a wave or set the value of a variable. 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, 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 are also callbacks that allow 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 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, virtually all XOPs will be resident.

The Basic Structure of an XOP

In very rough terms, you can think of your XOP program as shown in the following sketch which is best read from the bottom (high-level routines starting from XOPMain) to the top (low-level routines):

```
// Igor calls this directly for the corresponding external function
ExternalFunction()
{
    Process parameters
    Return result
}
// Igor calls this directly for the corresponding external operation
ExternalOperation()
{
    Process parameters
    Return result
}
XOPEntry()      // Igor sends most messages here
{
    Extract message from IORecHandle
    switch (message)
        Respond to message (often includes a callback to Igor)
    }
}
XOPMain()      // 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:

Chapter 1 — Introduction to XOPs

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

#include "XOPStandardHeaders.h" // Include ANSI-C, Mac or Win headers,XOP headers
#include 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
{
}

extern "C" 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 menu item
    }
}

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

extern "C" 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;
    }
}

HOST_IMPORT int
XOPMain(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

    return EXIT_SUCCESS;
}
```

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

The `XOPInit(ioRecHandle)` and `SetXOPEntry(XOPEntry)` calls in `XOPMain` 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.

The `RegisterOperation` call tells Igor the address of the C function to be called for the external operation added by the XOP.

Preparing to Write an XOP

Before you write your XOP:

- Read the source code for `XOP1` or `XFUNC1` (both are very short)
- Do the Guided Tour in Chapter 2
- Read Chapter 3 to learn how to build a project in your development system
- Read Chapter 4 to learn how Igor and your XOP interact

Typically you write an XOP by cloning one of the sample XOPs and gradually modifying it. The cloning process is more than a simple folder duplication as illustrated in the guided tour.

Here are some other topics that you might need to know about after reading the chapters listed above.

Read **Data Sharing** on page 65 and **Macintosh Memory Management** on page 197 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 57.

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 203.

If your XOP saves its settings or documents as part of an Igor experiment, read **XOPs and Experiments** on page 61.

To create a help file for your XOP, read Chapter 13, **Providing Help**.

Chapter 14, **Debugging**, discusses 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:

Chapter 1 — Introduction to XOPs

- The list of common pitfalls in Chapter 14
- 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 version of the operating system that you are using.
- 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.

World-Wide Web

Our Web address is www.wavemetrics.com.

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/support/mlist.htm>

IgorExchange

IgorExchange is a user-to-user support and collaboration site:

<http://www.IgorExchange.com>

If you create an XOP you might want to post it on IgorExchange for use by other Igor users.

Chapter 2

Guided Tour

Overview

This manual describes version 6.40 of the Igor XOP Toolkit. It focuses on development using Xcode 6 on Macintosh and Visual C++ 2015 on Windows. We removed instructions for older development systems from this version of the manual. If you must use an older development system, download the XOP Toolkit 6.30 manual from:

<ftp://ftp.wavemetrics.net/IgorXOPToolkit/Manual/Old/XOPToolkitManual630.pdf>

XOP Toolkit 6.40 creates XOPs that run with Igor Pro 6.20 or later.

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:

- Apple Xcode 6 on OS 10.9 or later
- Microsoft Visual C++ 2015 (including Express Edition)

If you are using a newer development system, you may need to make adjustments but the basic process remains the same.

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

1-Xcode 6.

<Step for Xcode 6>.

These steps also work in Xcode 7.

1-Visual C++ 2015.

<Step for Visual C++ 2015>.

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

In this guided tour you must use the 32-bit version of Igor, not the 64-bit version.

Activating XOPs

In the guided tour, you will be instructed to activate an XOP multiple times. Prior to Igor Pro 6.1, activating meant putting the XOP or an alias (*Macintosh*) or shortcut (*Windows*) pointing to it in the "Igor Extensions" folder inside the "Igor Pro Folder".

Chapter 2 — Guided Tour

In Igor Pro 6.1 or later, the recommended way to activate an XOP is to use the “Igor Extensions” folder inside “Igor Pro User Files” folder. If you are not familiar with this type of activation and the “Igor Pro User Files” concept, please read **Activating XOPs Using the Igor Extensions Folder** on page 5 now.

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 IgorXOPs6 folder and contains the following project files:

Xcode6/SimpleFit.xcodeproj	For Xcode 6. The compiled XOP will run on Mac OS X 10.6 or later.
VC2015\SimpleFit.sln	For Visual C++ 2015 on Windows. The compiled XOP will run on Windows Vista or later.

Building SimpleFit in Xcode 6

1-Xcode 6.

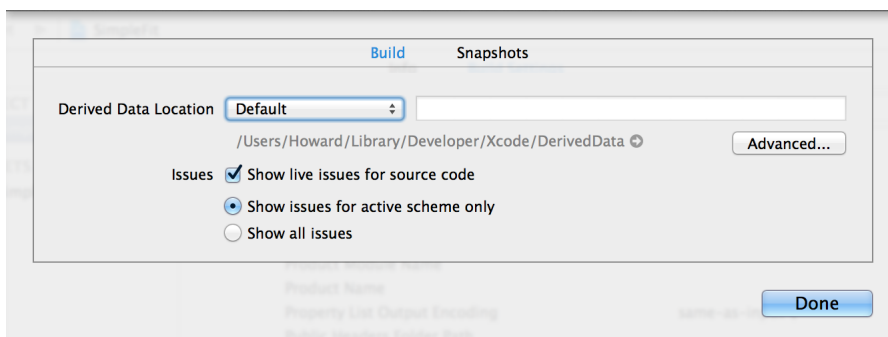
Open the SimpleFit/Xcode6/SimpleFit.xcodeproj project file in Xcode 6.

The SimpleFit project window is shown.

Xcode 6 may show a warning icon. If you click the warning icon, it may tell you about “updating to recommended settings”. Do not perform any updates as they may cause compile-time or run-time issues.

2-Xcode 6

Choose File→Project Settings to display the Project Settings dialog:

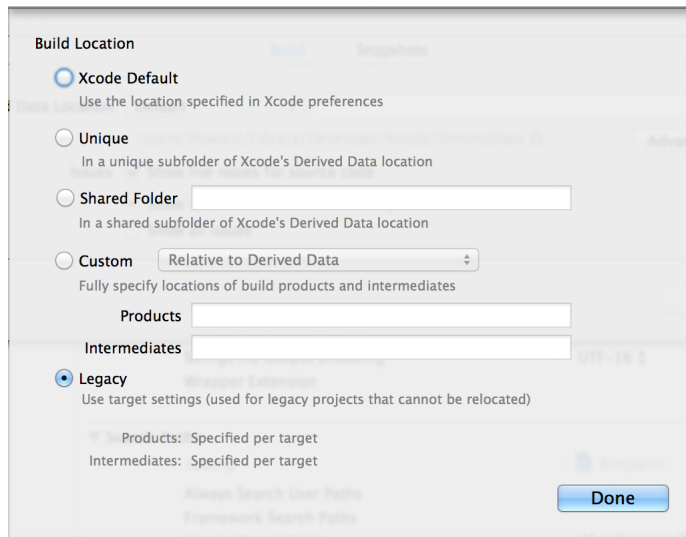


Although this dialog is called “Project Settings”, Xcode 6 stores these settings on a per-user basis. Consequently they must be set separately by each user and can not be preset in the XOP Toolkit sample XOPs.

NOTE: If you open other sample XOP Xcode projects in the future, or open this project while running

as a different user, you need to remember to set these project settings.

Click the **Advanced** button to display the **Advanced Project Settings** dialog:



Click the **Legacy** radio button.


This causes the compiled XOP to be stored where we expect it, in the SimpleFit/Xcode6/Debug folder.

NOTE: If you do not set the build location to Legacy your compiled XOP will be created in a strange location such as ~Developer/Xcode/Derived Data. So remember to set this setting for each XOP project you open in Xcode.

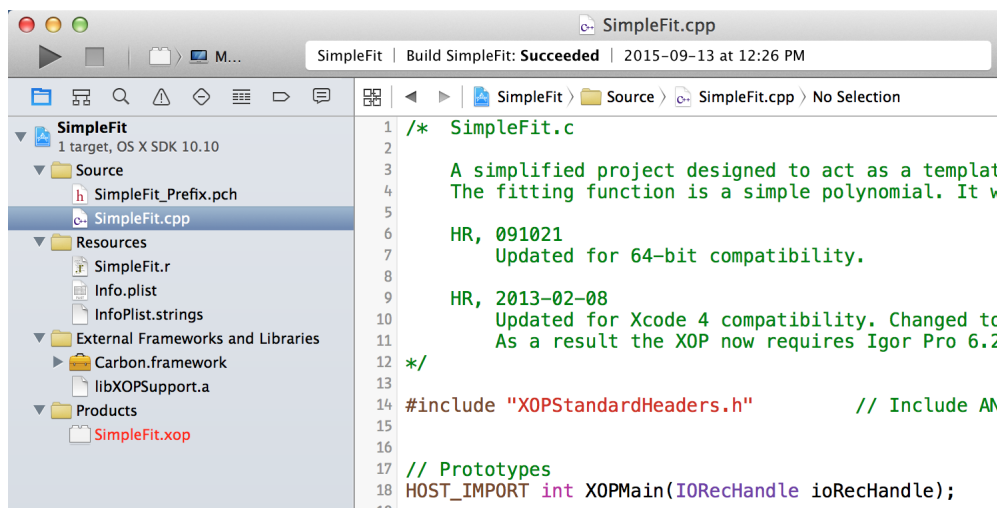
Click **Done** to close the **Advanced Project Settings** dialog.

Click **Done** to close the **Project Settings** dialog.

3-Xcode 6.

Click the **Project Navigator** icon () and open the disclosure triangles in the project navigator pane and select the **SimpleFit.cpp** file.

The project window should now look like this:



4-Xcode 6.

Choose **Product**→**Build**.

This is the same as choosing **Product**→**Build For**→**Build For Running**.

Chapter 2 — Guided Tour

In a few seconds the compiler will finish and an XOP package named SimpleFit.xop will be created in your SimpleFit/Xcode6/build/Debug folder. Note where this is located for use in an upcoming step.

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

5-Xcode 6.

Close the project window.

Building SimpleFit in Visual C++ 2015

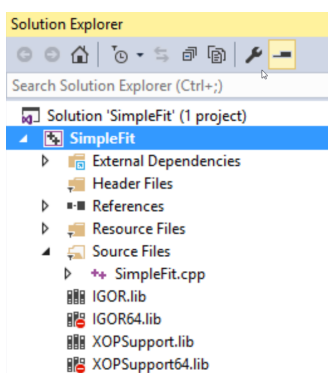
1-Visual C++ 2015.

Open the solution file in SimpleFit\VC2015\SimpleFit.sln file in Visual C++ 2015.

The SimpleFit project window is shown.

Choose View→Solution Explorer and open the SimpleFit icon.

The window should now look like this:



2-Visual C++ 2015.

Choose Build→Build Solution.

In a few seconds the compiler will finish and an XOP file named SimpleFit.xop will be created in your SimpleFit\VC2015 folder. Note where this is located for use in an upcoming step.

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

3-Visual C++ 2015.

Choose File→Close Solution.

Testing SimpleFit

Now that we have built the SimpleFit XOP, our next task is to test it.

In the next step, we activate the SimpleFit XOP so that Igor will load it. For a detailed explanation of activating, see **Activating XOPs Using the Igor Extensions Folder** on page 5.

1. **On the desktop, make an alias (Macintosh) or shortcut (Windows) for the newly created SimpleFit.xop and drag the alias or shortcut into the Igor Extensions 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.

2. **Back in the SimpleFit folder, make an alias (Macintosh) or shortcut (Windows) for the "SimpleFit Help.ihf" file and drag 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. Igor will find the help file if you put an identically-named alias or shortcut to it in the same folder as the XOP.

3. Launch Igor Pro.

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

4. Choose Command Help from the Help menu.**5. Select External from the Functions popup menu above the list and uncheck the Operations and Programming checkboxes.****6. 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.

7. 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*) but this does not work in the Igor Help Browser.

8. Quit Igor without saving.

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.

Cloning a project entails cloning the project folder and its contents: source files, resource files, help files and project files. In this case, here are the files:

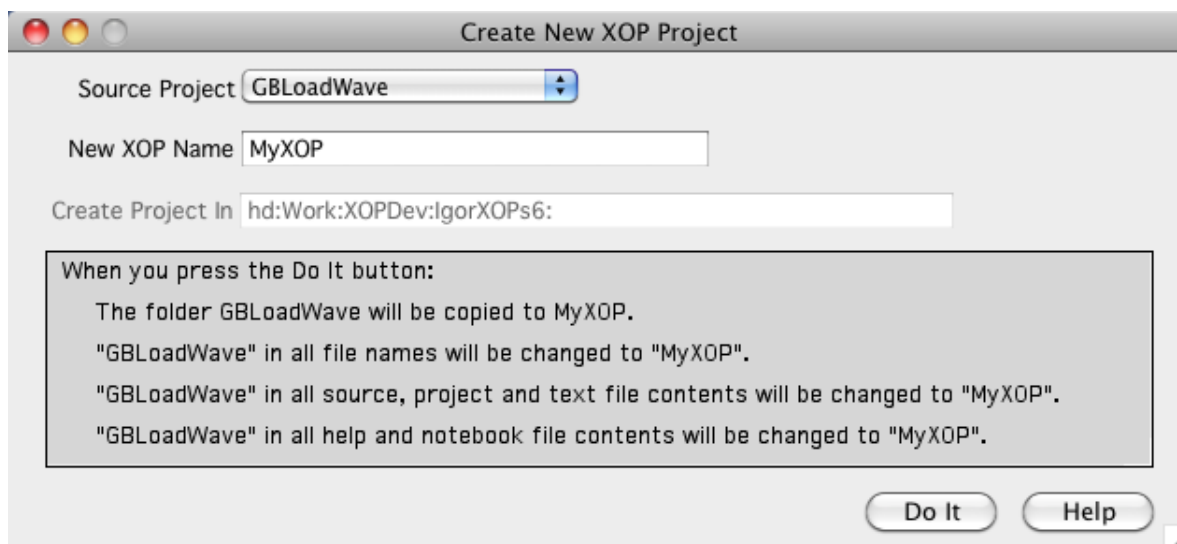
SimpleFit File	SimpleGaussFit File	Description
SimpleFit.cpp	SimpleGaussFit.cpp	C++ file containing XOP source code
SimpleFit.r	SimpleGaussFit.r	Macintosh resource file
SimpleFit.rc	SimpleGaussFit.rc	Windows resource file
SimpleFitWinCustom.rc	SimpleGaussFitWinCustom.rc	Windows resource file containing XOP Toolkit-specific resources
resource.h	resource.h	Windows resource include file
SimpleFit Help.ihf	SimpleGaussFit Help.ihf	Igor help file
SimpleFit.xcodeproj	SimpleGaussFit.xcodeproj	Xcode project
SimpleFit.sln	SimpleGaussFit.sln	Visual C++ solution file
SimpleFit.vcxproj	SimpleGaussFit.vcxproj	Visual C++ project file
SimpleFit.vcproj.filters	SimpleGaussFit.vcproj.filters	Visual C++ project file

We use the “Create New XOP Project” Igor experiment, which you can find at the top level of the IgorXOPs6 folder, to clone a project.

NOTE: The following steps require Igor Pro 6.10 or later. If you are using an older version of Igor, please update your Igor Pro installation now by choosing Igor’s Help→Updates For Igor menu item.

1. On the desktop, locate your IgorXOPs6 folder.**2. Open the “Create New XOP Project.pxp” file in Igor.**

The experiment opens and the Create New XOP Project panel appears:



3. From the Source Project popup menu, choose SimpleFit.

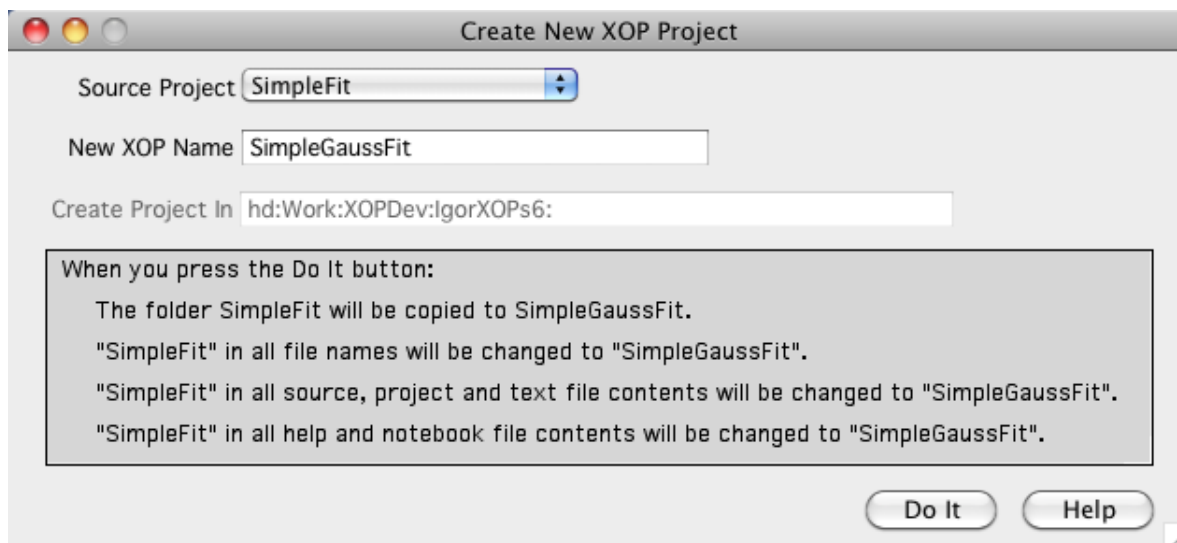
4. In the New XOP Name textbox, enter SimpleGaussFit.

Double-check your spelling to make sure you have entered SimpleGaussFit exactly.

Click in the panel outside of the New XOP Name textbox to finalize the setting.

5. Verify that the Create Project In textbox contains the path to your IgorXOPs6 folder.

The panel should now look like this, though your path will be different. Note the text in the status area which explains what will happen when you click Do It.



6. Click the Do It button.

Create New XOP Project clones the SimpleFit folder, creating the SimpleGaussFit folder.

In the clone, "SimpleFit" in all file names is changed to "SimpleGaussFit" and the contents of all source files, project files, text files, notebooks and help files are changed to replace "SimpleFit" with "SimpleGaussFit".

7. Quit Igor without saving.

8. On the desktop, open the SimpleGaussFit folder and familiarize yourself with its contents.

Building the New Project in Xcode 6

Before modifying the new project we will build it to verify that it was correctly created.

1-Xcode 6.

On the desktop, open the Xcode6 folder in the SimpleGaussFit folder.

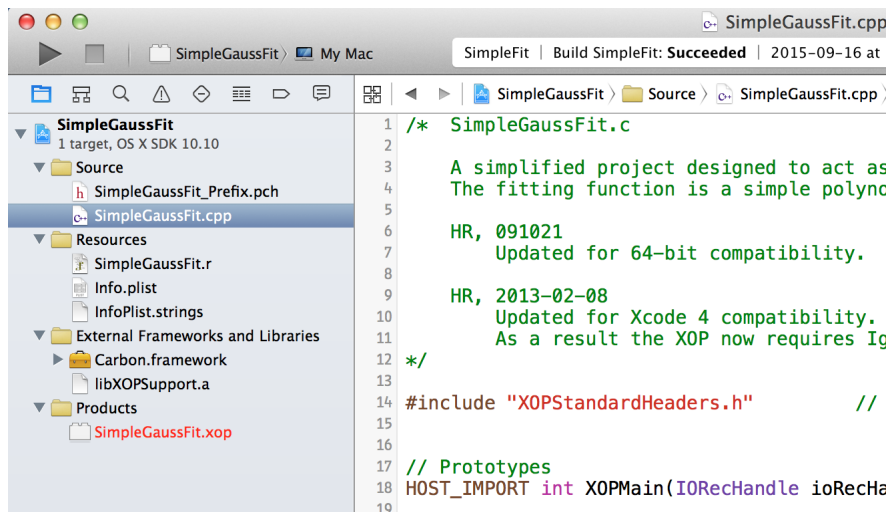
This folder contains files specific to the Xcode 6 development system.

They have also been tested in Xcode 7.01.

2-Xcode 6.

Open the SimpleGaussFit.xcodeproj file in Xcode 6.

This is what the project window looks like in Xcode 6 after opening the disclosure triangles and selecting the SimpleGaussFit.cpp file:



Xcode may show a warning icon. If you click the warning icon, it may tell you about “updating to recommended settings”. Do not perform any updates as they may cause compile-time or run-time issues.

3-Xcode 6.

If the icons in the SimpleGaussFit list are not open as shown above, click the Project Navigator icon (🗑️) and the disclosure triangles to open the groups.

Notice that the SimpleGaussFit source and resource files are referenced by the project as well as the Carbon and XOPSupport libraries.

Xcode project settings are discussed under **XOPs in Xcode** on page 31. You don’t need to know about them for this guided tour.

4-Xcode 6.

Choose File→Project Settings to display the Project Settings dialog.

5-Xcode 6.

Click the Advanced button to display the Advanced Project Settings dialog and then click the Legacy button if it is not already selected.

NOTE: This step must be done any time you open a sample XOP project or your own XOP project if it was not previously opened in Xcode on your machine or if you open it as a different user.

6-Xcode 6.

Click Done to close the Advanced Project Settings dialog.

7-Xcode 6.

Chapter 2 — Guided Tour

Click Done to close the Project Settings dialog.

8-Xcode 6.

Choose Product→Build.

This is the same as choosing Product→Build For→Build For Running.

Xcode will build the project. You may get some warnings. When Xcode is finished, it should display a “Succeeded” message in the Activity Viewer section (top/center) of the Xcode main window.

9-Xcode 6.

On the desktop, verify that you have a package folder named SimpleGaussFit.xop in IgorXOPs6/SimpleGaussFit/Xcode6/build/Debug.

Debug is the name of the Xcode configuration currently being built so the output goes into the /build/Debug directory.

The SimpleGaussFit.xop 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. Inside the package you will find a file named SimpleGaussFit. This is the executable file and contains Intel code only because Xcode 6 does not compile for PowerPC.

10-Xcode 6.

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

We now have a SimpleGaussFit XOP but it behaves just like the SimpleFit XOP. We will fix that in the following sections.

Building the New Project In Visual C++ 2015

1-Visual C++ 2015.

On the desktop, open the VC2015 folder in the SimpleGaussFit folder.

This folder contains files specific to the Visual C++ development system.

2-Visual C++ 2015.

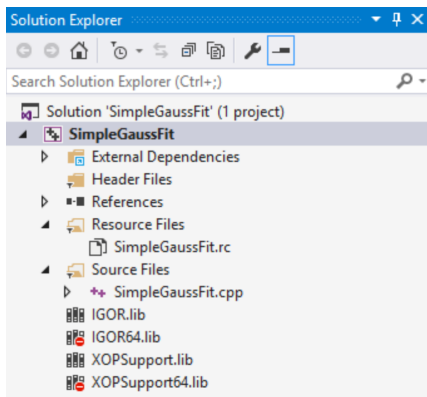
Open the SimpleGaussFit.sln file in Visual C++ 2015.

The SimpleGaussFit project window is shown.

3-Visual C++ 2015.

If the icons in the Solution Explorer window are not open as shown above, click the icons to open them.

This is what it looks like in VC 2015:



Notice that the source and resource files are referenced by the project as well as the 32-bit and 64-bit IGOR and XOPSupport libraries.

Visual C++ project settings are discussed under **XOPs in Visual C++ 2015** on page 43. You don't need to know about them for this guided tour.

4-Visual C++ 2015.

Choose Build→Build Solution.

Visual C++ should build the XOP with no errors (though you may get some warnings), creating the file SimpleGaussFit.xop in your IgorXOPs6\SimpleGaussFit\VC2015 folder.

The warnings about XOPSupport.lib, if any, are generated because you are linking a non-debug library to the debug version of your XOP. This is not a problem.

5-Visual C++ 2015.

On the desktop, verify that you have a file named SimpleGaussFit.xop in IgorXOPs6/SimpleGaussFit/VC2015.

This is the compiled XOP file.

6-Visual C++ 2015.

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

We now have a SimpleGaussFit XOP but it behaves just like the SimpleFit XOP. We will fix that in the following sections.

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.

The "SimpleGaussFit Help.ihf" file was created when the Create New XOP Project experiment cloned the "SimpleFit Help.ihf" file. The Create New XOP Project control panel replaced all occurrences in the help file of "SimpleFit" with "SimpleGaussFit". We now need to finish modifying "SimpleGaussFit Help.ihf".

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

Here is some suggested body text:

SimpleGaussFit is a curve 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.

3. **Save the notebook by choosing File→Save Notebook.**
4. **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.
5. **Choose File→Open→Help File and open the "SimpleGaussFit Help.ihf" file as a help file.**
Igor will display a 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.

6. Kill 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.

7. Quit Igor without saving.

Changing and Compiling the Code

The SimpleGaussFit.cpp file was created when the Create New XOP Project experiment cloned the SimpleFit.cpp file. Create New XOP Project replaced all occurrences in the help file of “SimpleFit” with “SimpleGaussFit”.

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.cpp for editing.
2. 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.

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

You can skip this step but it is what you should do when creating a real XOP.

4. **Save your changes to the SimpleGaussFit.cpp file.**

5-Xcode 6.

Compile the project by choosing Product→Build.

The compile and link should proceed without errors (although you may get some warnings), creating SimpleGaussFit.xop in the IgorXOPs6/SimpleGaussFit/Xcode6/build/Debug folder. If you get errors, carefully check the changes that you made to the source code.

5-Visual C++ 2015.

Compile the project by choosing Build→Build Solution.

The compile and link should proceed without errors (although you may get some warnings), creating SimpleGaussFit.xop in the IgorXOPs6\SimpleGaussFit\VC2015 folder. If you get 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. **If it is running, 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, make an alias (Macintosh) or shortcut (Windows) for the "SimpleGaussFit Help.ihf" file and drag the alias/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 (SimpleGaussFit.xop in this case). Normally the help file and the XOP reside in the same folder but this is not true during development. Igor will find the help file if you put an identically-named alias or shortcut to it in the same folder as the XOP.

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 in the Help Browser. Igor gets the help from the "SimpleGaussFit Help.ihf" file which it looks for in the same folder as the SimpleGaussFit XOP. The help file does not need to be open for Igor to get operation or function help from it.

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

- Verify that you created an alias (Macintosh) or shortcut (Windows) from SimpleGaussFit.xop and placed the alias or shortcut in the Igor Extensions folder. See **Activating XOPs Using the Igor Extensions Folder** on page 5 for details.
- If you have multiple copies of Igor on your machine, make sure that you are running the copy you intended to run.
- Make sure that you are running the 32-bit version of Igor, not the 64-bit version.

Chapter 2 — Guided Tour

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

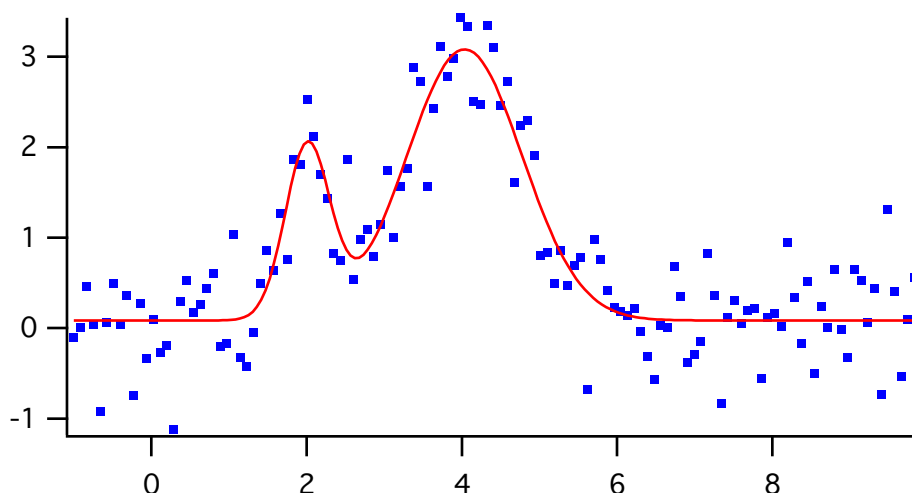
- Verify that you created an alias (*Macintosh*) or shortcut (*Windows*) from the “SimpleGaussFit Help.ihf” help file and placed the alias or shortcut in the folder containing the XOP. The alias or shortcut must have the same name as the help file itself.
- Verify that you compiled the “SimpleGaussFit Help.ihf” file as directed in **Changing the Help** on page 25.

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)/1.0)^2         // second peak
Display data
ModifyGraph 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
```

The result should look something like this:



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

To avoid errors due to arithmetic truncation it is best to use double-precision for curve fitting so now we will redimension to double-precision and redo the fit.

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, carefully check the changes that you made to SimpleGaussFit.cpp 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.

Touring the Code

In this section we take a very brief tour of the code to give you a sense of how an XOP works.

The details are explained in subsequent chapters. For now just try to get the big picture.

1. **In your development system, open the SimpleGaussFit.cpp file and scroll to the bottom until you see the XOPMain function.**

In most sample XOP source files, the lower-level code appears at the top and the higher-level code at the bottom. It is usually easier to understand higher-level code, so we examine the higher-level code first.

The XOPMain function for all XOPs is similar.

XOPInit allows the XOPSupport library to initialize itself.

SetXOPEntry tells Igor what function to call to pass messages to the XOP.

Next comes a check to make sure that the current Igor version is not too old. XOPs created in XOP Toolkit 6.3 or later require Igor Pro 6.20 or later so every XOP should contain this check.

SetXOPResult tells Igor the result of the call to the XOP's XOPMain function. 0 means "success".

The function result (EXIT_FAILURE or EXIT_SUCCESS) is currently not used by Igor but may be used by a future version.

If you had additional initialization to do, you would add it after the check of the Igor version. In XOPs that add an external operation, you will see a call to RegisterOperation at this point. SimpleGaussFit adds no external operations, just an external function.

2. **Scroll up until you see the XOPEntry function.**

The SetXOPEntry call in the XOPMain function told Igor to pass subsequent messages to the XOPEntry function.

The XOPEntry routine receives messages such as MENUITEM, IDLE, CLEANUP and many others from Igor. They are listed under **Basic XOP Messages** on page 54 but you don't need to know about that for now. XOPs can ignore messages that they don't care about. Most XOPs respond to just a few messages.

XOPEntry has no function parameters because the message and its parameters are passed in a global handle. The function returns void because the XOP sends the result back to Igor via the same global handle by calling SetXOPResult. You don't access the global handle directly - the XOPSupport library accesses it for you when you call XOPSupport routines like SetXOPEntry, GetXOPMessage and SetXOPResult.

Igor does not send messages to XOPEntry for external operation and external function calls but instead calls an XOP-supplied function directly.

If the XOP defines an external function, as SimpleGaussFit does, during initialization Igor sends the FUNCADDRESS message once for each external function. The XOP responds by returning the address of the XOP function to call when that external function is invoked from an Igor command. In SimpleGaussFit, this is done with a call to RegisterFunction, defined just above XOPEntry.

3. **Scroll up until you see the SimpleGaussFit function.**

This is the function that Igor calls when the SimpleGaussFit external function is invoked from an Igor command. Parameters to the external function are passed via a structure. This is explained in Chapter 6, **Adding Functions**.

Next we will examine the resources that tell Igor what the XOP adds.

At launch time, Igor examines each XOP's resources to determine what the XOP adds to Igor. Most XOPs add one or more external operations or external functions, or both. Most XOPs add error messages and some add menu items.

4-Xcode 6.

Open the SimpleGaussFit.r file.

The .r file on Macintosh contains standard Macintosh resources (e.g., vers, STR#) and XOP-specific resources (e.g., XOPC, XOPF).

4-Visual C++ 2015.

Open the SimpleGaussFit.rc file for editing as a text file by right-clicking it in Solution Explorer and choosing View Code.

The .rc file that appears in the Solution Explorer contains standard Windows resources (e.g., VERSION-INFO) only. XOP-specific resources (e.g., XOPC, XOPF) are stored in a separate .rc file that is referenced from the main .rc file.

Scroll to the bottom of the SimpleGaussFit.rc file until you see this line:

```
"#include "SimpleGaussFitWinCustom.rc" "\r\n"
```

SimpleGaussFitWinCustom.rc contains the XOP-specific resources. They are what we are interested in at this point. We will now open SimpleGaussFitWinCustom.rc for editing as a text file.

Choose File→Open→File.

Navigate to the SimpleGaussFit folder.

Select SimpleGaussFitWinCustom.rc in the file list.

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

Select “Source Code (Text) Editor”.

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

5. Scroll down until you see the XOPF resource.

This resource tells Igor what external functions the XOP adds to Igor. In the case of SimpleGaussFit, just one external function is added. XOPs that add external operations would have an XOPC resource that serves a similar purpose.

6. Scroll up until you see the XOPI resource.

This resource provides general information about the XOP to Igor. You do not need to change it.

7. Scroll up until you see the STR# 1100 resource.

XOPs can define their own error codes. This resource provides the error strings for XOP error codes.

More information about XOP resources is provided under **XOP Resources** on page 51.

Where To Go From Here

Read Chapter 3, **Development Systems** and Chapter 4, **Igor/XOP Interactions** in this manual.

If you plan to create an external operation, read Chapter 5, **Adding Operations**.

If you plan to create an external function, read Chapter 6, **Adding Functions**.

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

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

Chapter 3

Development Systems

Overview

XOP Toolkit 6.40 creates XOPs that run with Igor Pro 6.20 or later.

It includes sample XOPs and supporting files for the following development systems:

Development System	XOP Runs On	With Igor Version
Xcode 6.1 through 6.4	Mac OS X 10.6 or later	Igor Pro 6.20 or later
OS X 10.9 and 10.10	Intel processors only	
Visual C++ 2015	Windows 7 or later	Igor Pro 6.20 or later

On Macintosh, the sample projects were successfully tested with Xcode 7.01.

The sample projects will most-likely work with newer development system versions.

On Windows, the free Express Edition of Visual C++ is sufficient for most XOP development.

XOP Toolkit 6.4 includes project files for older development systems but this manual does not discuss them. If you must use an older development system, download the XOP Toolkit 6.30 manual from:

<ftp://ftp.wavemetrics.net/IgorXOPToolkit/Manual/Old/XOPToolkitManual630.pdf>

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.

The latest version of the XOP Toolkit is available for download from the same URL from which you originally downloaded the toolkit.

XOPs in Xcode

Xcode is Apple's development system for Mac OS X. It is available for download from the App Store and from the Apple Developer program web site.

XOP Toolkit 6.4 supports development using Xcode 6.1 through 6.4. It was also successfully tested with Xcode 7.01. It will likely work with later versions of Xcode.

Each sample project contains an Xcode6 folder. The Xcode6 folder, e.g., WaveAccess/Xcode6, contains the Xcode project file, e.g., WaveAccess.xcodeproj.

Chapter 3 — Development Systems

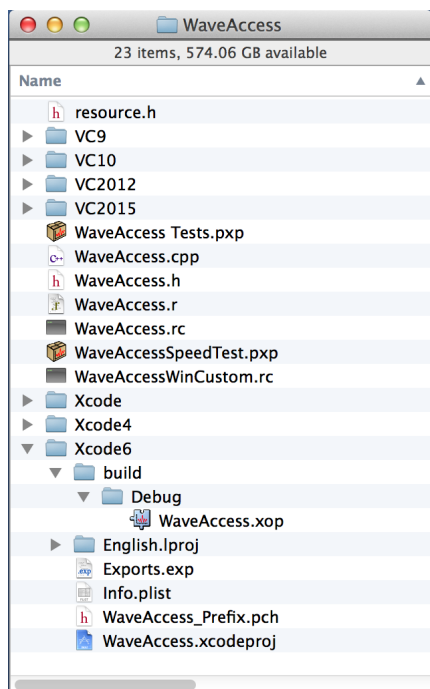
This table summarizes the relationship between Xcode versions and supported operating systems:

Development System	Xcode Runs On	XOPs Run On	Project Files Located In
Xcode 6.1 through 6.4	OS X 10.9	Mac OS X 10.6 or later	<project>/Xcode6
	OS X 10.10	Intel only	

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 21.

We will use the WaveAccess sample XOP as a case in point. The WaveAccess folder is inside the IgorXOPs6 folder and looks like this:



The Xcode project files are in the Xcode6 folder. The discussion assumes this arrangement and we recommend that you use it. Compiled XOPs are built in Xcode6/build/Debug.

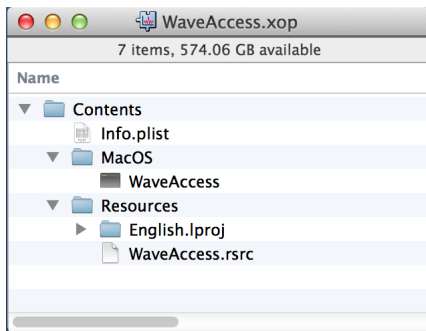
WaveAccess.xcodeproj is the project file and contains the project settings. (Actually it is a package folder, not a file.) The build folder contains data created by the compiler such as object code as well as the compiled XOP.

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

WaveAccess.r contains the XOP's Macintosh resources. The use of this resource file is further explained under **XOP Resources** on page 51. 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 XOPMain.

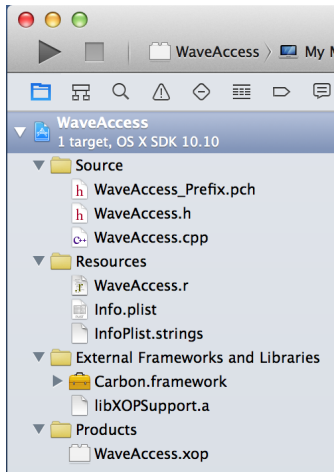
WaveAccess.xop appears to be a file but it is really a “package”. If you control-click and choose Show Package Contents, you see what is inside:



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 Xcode 6 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 when the project is created. They store “meta-information” such as the XOP name of the executable file within the package folder.

The file `Carbon.framework` makes Apple’s Carbon API accessible to the XOP.

`libXOPSupport.a` is the WaveMetrics XOPSupport library.

Xcode 3 Project Settings

This version of the manual describes Macintosh XOP development with Xcode 6 only. For details on using Xcode 3, download the XOP Toolkit 6.30 manual from:

<ftp://ftp.wavemetrics.net/IgorXOPToolkit/Manual/Old/XOPToolkitManual630.pdf>

Xcode 4 Project Settings

This version of the manual describes Macintosh XOP development with Xcode 6 only. For details on using

Xcode 4, download the XOP Toolkit 6.30 manual from:

<ftp://ftp.wavemetrics.net/IgorXOPToolkit/Manual/Old/XOPToolkitManual630.pdf>

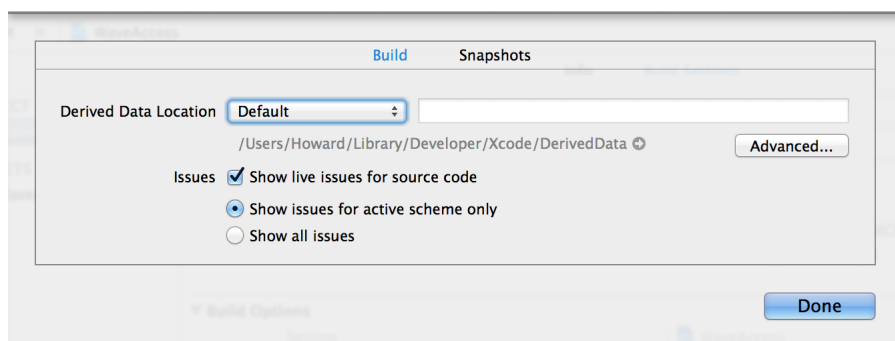
Xcode 6 Project Settings

In Xcode 6, settings can be made at the project, target or configuration level. If no value is set for a given setting, it inherits the Xcode default value. For most settings it does not much matter whether they are set at the project level or at the target level.

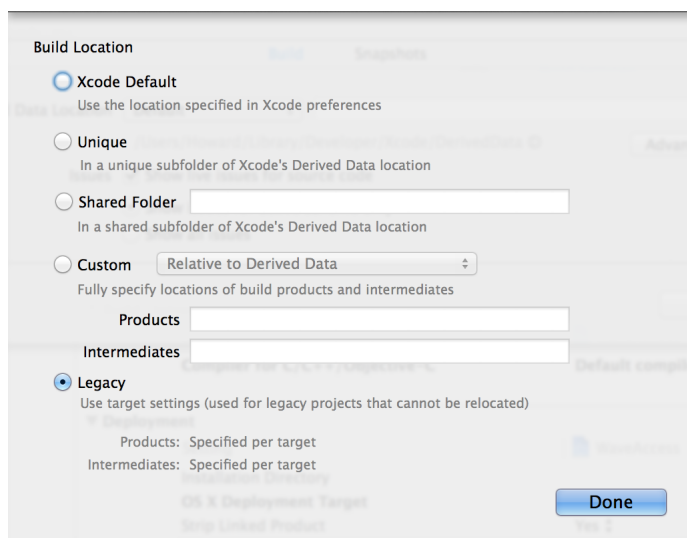
Xcode has two additional kinds of settings: “project settings” and “schemes”. Confusing though it is, *project settings* are set via the File→Project Settings dialog while project *build* settings are set via the project navigator.

Xcode 6 Project Settings Dialog

You access the Xcode 6 Project Settings dialog by choosing File→Project Settings:



If you click the Advanced button, you get the following subdialog:



The *Build Location* setting should be set to “Legacy”. This causes the compiled XOP to be stored where we expect it, in the <project>/Xcode6/Debug folder.

Xcode stores these settings on a per-user basis. Project settings set by one user are not accessible by another user. Consequently they can not be preset by WaveMetrics and must be set separately by each user.

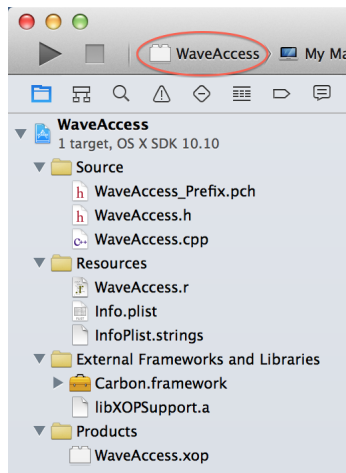
NOTE: If you do not set the build location to Legacy your compiled XOP will be created in a strange location such as ~Developer/Xcode/Derived Data. So remember to set this setting for each XOP project you open in Xcode.

NOTE: If you open any sample XOP Xcode project or open a project while running as a different user, you need to remember to set these project settings.

Alternatively, you can set these settings using Xcode preferences in which case they will be applied to any Xcode project that you open in Xcode for the first time as a given user. To do this, choose Xcode→Preferences, click the Locations icon, click the Advanced button, and click the Legacy radio button.

Xcode 6 Schemes

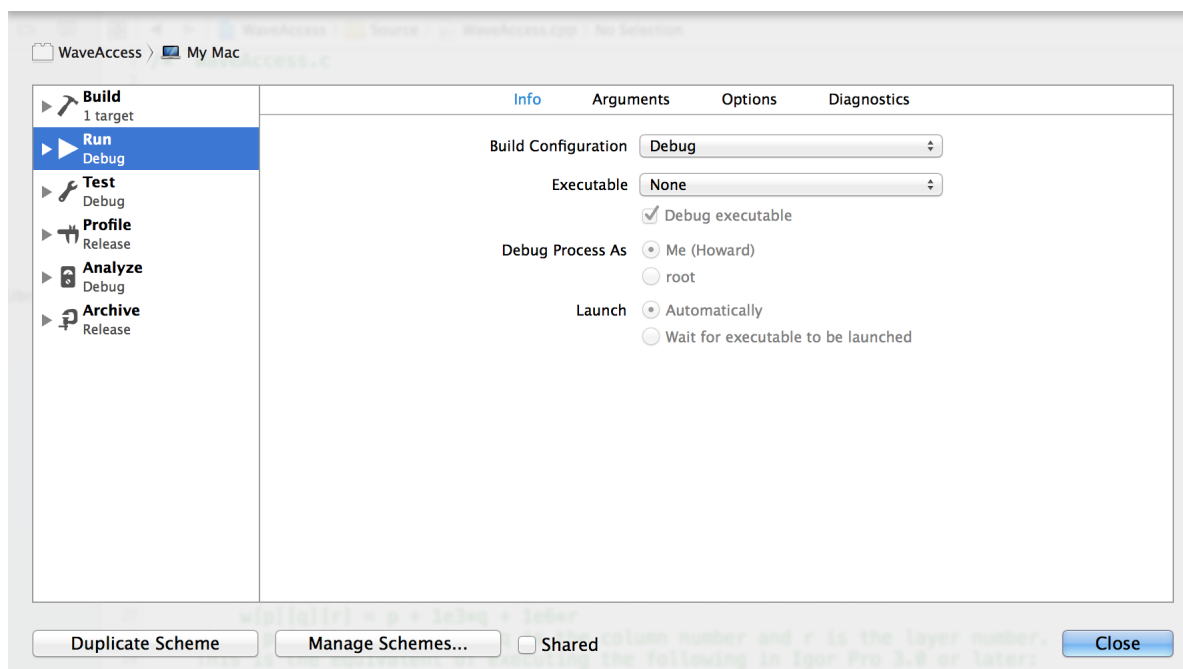
An Xcode *scheme* is a collection of settings that control things like what configuration (debug or release) is built and which application to run when you ask to debug the program. You choose which scheme to use through the scheme popup menu in the top/left corner of the Xcode main window. Here the scheme name is shown as WaveAccess:



The popup menu is really two popup menus in one. If you click “WaveAccess”, you can select or edit the scheme. If you click “My Mac 32-bit” you can select the “destination”. For Igor XOPs the destination is always 32-bit Macintosh.

NOTE: If you see “My Mac 64-bit” instead of “My Mac 32-bit” it is likely that the project is set to use an SDK (software development kit) that you do not have on your machine (e.g., “Mac OS X 10.6 SDK”). Fix this in the project build settings by choosing an available SDK as described under **Xcode SDKs** on page 40.

If you click the popup menu and choose Edit Scheme, you see this:



Xcode stores schemes on a per-user basis. A scheme created by one user is not accessible by another user. Consequently they can not be preset by WaveMetrics and must be set separately by each user.

The sample XOP projects ship with no schemes. When you first open a sample XOP project, Xcode creates a default scheme such as “WaveAccess - My Mac 32-bit” shown above. This scheme is set up to use the debug configuration.

For the most part that’s all you need to know about schemes. You can skip the rest of this section unless you have a problem with schemes.

Under some circumstances, Xcode will not create the default scheme and you will have to intervene. This appears to be a design flaw of Xcode and may be fixed in a later version.

If you open a project as a given user, say User1, Xcode will create a default scheme. If you then open the project as a different user, say User2, Xcode will not create a default scheme and the scheme popup menu will say No Scheme. If you click New Scheme and try to create a new scheme, Xcode will not create a scheme and you will be unable to build the project.

The reason for this problem has to do with permissions. When User1 opened the project for the first time, Xcode created folders within the WaveAccess.xcodeproj package and set these folders to be read-only for everyone except User1. When User2 tries to create a scheme, he can not because he has only read-only permission for these internal folders.

The solution is to give everyone read-write access to the Xcode project file and everything in it. You must be running as an administrator to do this.

Run Terminal and execute these commands, substituting your actual path for <path to your IgorXOPs6 folder>:

```
cd <path to your IgorXOPs6 folder>
chmod -R 0777 WaveAccess/Xcode6/WaveAccess.xcodeproj
```

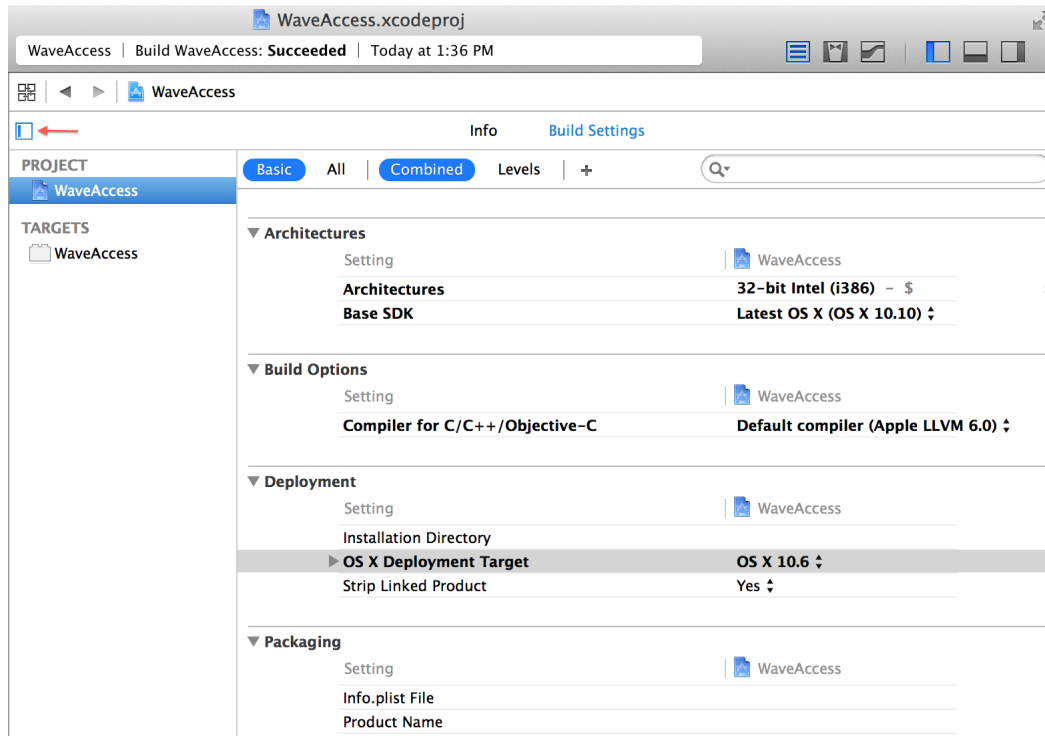
If you now open the project as any user, the default scheme will be created.

Xcode 6 Project Build Settings

To display *project build* settings in Xcode, choose View→Navigators→Show Project Navigator and click the WaveAccess icon at the top of the project navigator. The project editor pane appears in the middle of the main Xcode window.

If you don't see the labels PROJECT and TARGETS, as shown below, click the Projects and Targets List button. This is the mysterious little rectangular icon with the solid little vertical rectangle running down its left side. In the screen dump below, we have added a red arrow pointing to this icon.

Click WaveAccess under the PROJECT heading. Click the Build Settings tab and the Basic and Combined buttons:



Architectures: Set to “32-bit Intel”.

Base SDK: Set to Latest OS X.

C/C++ Compiler Version: Set to “Default compiler”. This selects the LLVM compiler.

Mac OS X Deployment Target: Set to 10.6. If you really need to run on OS X 10.4, you can try setting it to 10.4.

By setting the base SDK to “Latest OS X”, we guarantee that the project will work if you open it in a new OS version years from now. However this also lets you call Apple routines that are not available on the OS version that you happen to be running on, in which case your XOP will crash. For most XOPs, this is not a problem, because the XOPs use no Apple routines at all directly. However, if you are writing a platform-specific XOP that uses relatively new Apple routines, you may want to set the base SDK to the specific OS version that your XOP requires.

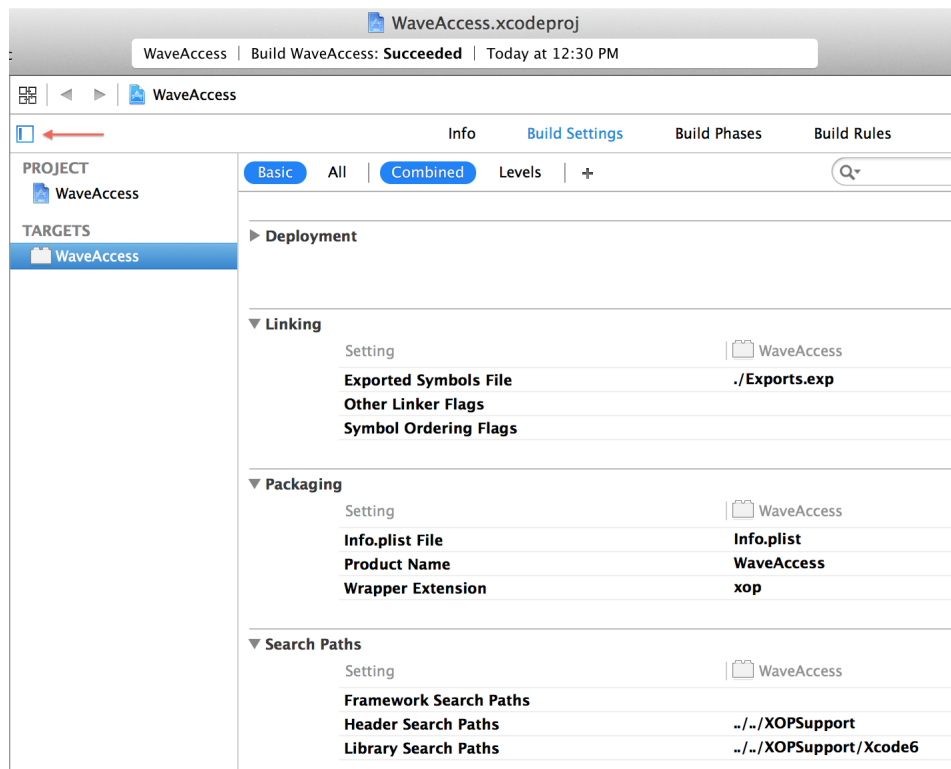
Xcode 6 Target Settings

To display *target* build settings in Xcode, choose View→Navigators→Show Project Navigator and click the WaveAccess icon at the top of the project navigator. The project editor pane appears in the middle of the main Xcode window.

Chapter 3 — Development Systems

If you don't see the labels PROJECT and TARGETS, as shown below, click the Projects and Targets List button. This is the mysterious little rectangular icon with the solid little vertical rectangle running down its left side. In the screen dump below, we have added a red arrow pointing to this icon.

Click WaveAccess under the TARGETS heading. Click the Build Settings tab and the Basic and Combined buttons:



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 XOPMain function to be exported (i.e., visible to Igor Pro by name). This file will be the same for all XOPs.

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

Header Search Paths: Allows #include statements to find the XOPSupport header files. The path is relative to the folder containing the .xcodeproj project file.

Library Search Paths: Specified as “../XOPSupport/Xcode6”. Allows the Xcode 6 version of the XOPSupport library to be found.

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

Deprecated Functions: We turn this warning off. You can turn it on if you decide to update your code to use all of the latest functions.

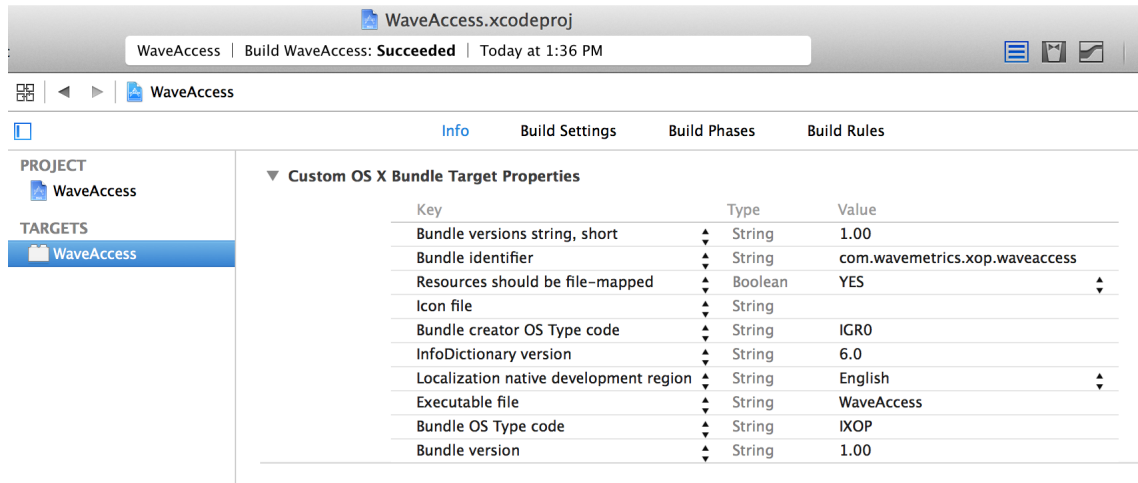
Other Warning Flags: Set to “-Wno-parentheses”. This turns off warnings for assigning the result of a function call to a variable in a conditional statement, such as:

```
if (err = SomeFunction())
```

OTHER_REZFLAGS: Set to “-i ../../XOPSupport”. This specifies flags to pass to the Rez compiler when the .r file is compiled. It allows #include statements to find the XOPStandardHeaders.r file in the XOPSupport folder.

Xcode 6 Target Info

To display *target info* settings in Xcode, choose View→Navigators→Show Project Navigator and click the WaveAccess icon at the top of the project navigator. The project editor pane appears in the middle of the main Xcode window. Click WaveAccess under the TARGETS heading. Click the Info tab:



These settings are written to the project’s Info.plist file which is copied to the XOP package’s Info.plist file when the XOP is compiled.

The values in Info.plist can be superseded by the InfoPlist.strings file. If values that you enter in the Properties tab do not take effect when the XOP is compiled, check if Xcode has propagated your changes to the InfoPlist.strings file. If not, manually update the InfoPlist.strings file.

Executable File: Set to the name of the executable file in the XOP package. For example, for the WaveAccess XOP, the package name is WaveAccess.xop and the executable file, named WaveAccess, is located in the WaveAccess package at Contents/MacOS/WaveAccess.

When Igor loads your XOP, if you get an error such as “Can’t load the executable file in the XOP package”, you may have an inconsistency in your settings. The Executable File setting must match the target product name setting (under Packaging in the target build settings). The Executable File setting corresponds to the CFBundleExecutable XML key in the Info.plist file. This is explained further under **Check Your Info.plist File** on page 238.

Bundle Identifier: The bundle identifier is a unique string associated with an executable. By convention it uses Java package notation like com.wavemetrics.xop.waveaccess. For your own XOPs you can make up an identifier of your choice.

Bundle OS Type Code: Set to IXOP. This is necessary for the XOP package to be displayed in the Finder using the correct icon for an XOP.

Bundle Creator OS Type Code: Set to IGR0. This associates the XOP with the Igor Pro application.

Bundle Version and Bundle Version String: Set the version for the XOP package. This is the version reported by the Finder.

Xcode SDKs

An SDK (software development kit) is a set of libraries, header files and other components needed to build an executable on Mac OS X. Apple releases a new SDK with each operating system release. The new SDK adds support for new system routines and sometimes removes support for old system routines.

In each Xcode project you specify the SDK that you want to use via the Base SDK setting in the project build settings.

As Apple ships new versions of Xcode, it removes SDKs from older OS versions. You can find the SDKs included with your version of Xcode at:

```
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs
```

If you choose “Mac OS X 10.9” as the base SDK for your project you will be able to use system routines defined for Mac OS X 10.9 but not routines defined for Mac OS X 10.10 because these require the 10.10 SDK. However, there is another problem. If you select, for example, “Mac OS X 10.9” as the base SDK for your project, in a year or two, when you are using a new version of Xcode on a new version of OS X, your Xcode will not include the 10.9 SDK and your project will not compile. You can fix this by choosing a newer SDK.

In XOP Toolkit 6.40, we elected to set the Xcode 6 projects up to use the “Latest OS X” SDK. This prevents the problem described above where the project ceases to compile. But it introduces a new problem. It allows you to call Mac OS X routines added in the latest OS version when you are running on an older OS version. This will cause a crash.

This is not a serious problem for most XOPs because most XOPs are platform-independent and do not use any OS routines directly. However, if you are an advance programmer writing a platform-dependent XOP that uses new OS routines, you need to think about this issue so you do not call a new routine on an old OS.

As of the 10.7 SDK, Apple removed support for some old technologies such as Dialog Manager and QuickDraw. These are used by the GBLoadWave, VDT2 and WindowXOP1 XOPs. Consequently, we do not ship Xcode 6 projects for those XOPs. XOP Toolkit 7 addresses this issue by using newer techniques.

Missing SDK With Xcode 6

If you open an Xcode project that uses an SDK that is not included in your current Xcode, Xcode will display a warning saying that the SDK is missing. To fix this, set the project SDK to Latest Mac OS X as explained under **Xcode 6 Project Build Settings** on page 37.

Converting to the LLVM Compiler

Prior to XOP Toolkit 6.30 the sample XOP projects were configured to use the GCC compiler. Apple dropping support for the GCC compiler in favor of the LLVM compiler. Consequently in XOP Toolkit 6.30 all sample projects were converted to use LLVM.

If you are updating an old project, you will need to set it to use the LLVM compile as explained under **Xcode 6 Project Build Settings** on page 37.

You also may need to convert your XOP to use XOPMain as its main function. See **The XOPMain Function** on page 215 for details.

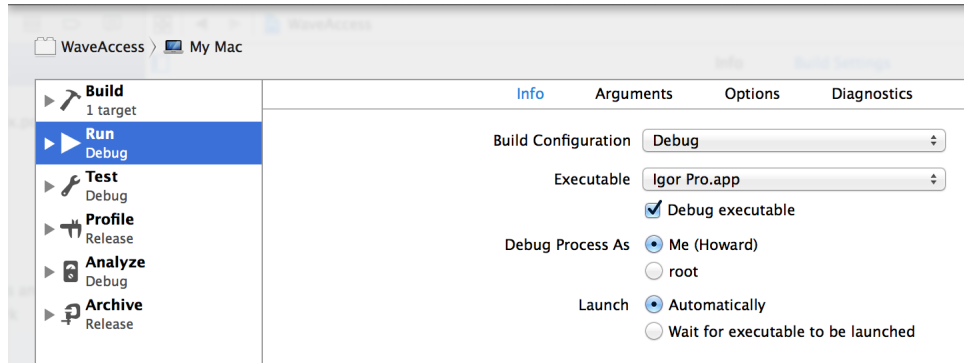
Debugging an XOP in Xcode 6

You need to tell Xcode what program hosts your XOP. Here's how you would do this after opening the WaveAccess sample project in Xcode:

1. Choose Product→Scheme→Edit Scheme.
2. Click Run in the list of actions on the left.

3. Click the Info tab.
4. Choose Debug from the Build Configuration popup menu.
5. Click the Executable popup menu, choose Other, select your Igor Pro application file and click the Choose button.

The scheme editor window should now look like this:



6. Click OK.
7. Choose Product→Build.
This is the same as choosing Product→Build For→Build For Running.
When the project compilation is finished, go into the Finder and find the compiled XOP (/WaveAccess/Xcode6/build/Debug/WaveAccess.xop).
8. Make an alias for that XOP and activate it as described under **Activating XOPs Using the Igor Extensions Folder** on page 5.
9. In Xcode, open the your main source file (e.g., WaveAccess.cpp) and set a breakpoint at the start of the XOPMain function. Click in the left-hand gutter to set the breakpoint.
10. In Xcode choose Product→Run.
Xcode will launch Igor Pro.
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 XOPMain function where you set the breakpoint.
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.

Converting an Old Xcode Project

This section documents the steps we took to convert an Xcode 3 project so that it could be compiled using Xcode 6.4.

Converting WaveAccess to Xcode 6

Here is the process we used to convert the Xcode 3 WaveAccess project to Xcode 6.

1. Duplicate the WaveAccess/Xcode folder and name the duplicate Xcode6.
2. Launch Xcode 6.
3. Open WaveAccess/Xcode6/WaveAccess.xcodeproj in Xcode 6.
4. Choose View→Navigators→Show Project Navigator.

The Project Navigator appears in the lefthand pane if it was not already visible.

5. Select the WaveAccess icon at the top of the Project Navigator pane.

6. Choose View→Utilities→Show File Inspector.

The File Inspector appears in the righthand pane if it was not already visible.

7. In the File Inspector, under Project Format, choose “Xcode 3.2-compatible”.

8. Notice the WaveAccess icon in the Project Navigator (lefthand pane).

Xcode shows a message below the project icon saying “1 target, missing base SDK”. This is because the XOP Toolkit sample Xcode 3 projects are set up to use the OS X 10.4 SDK but Xcode 6 does not include the 10.4 SDK.

9. Choose File→Project Settings.

10. Optionally uncheck the “Show live issues for source code” checkbox.

The Live Issues feature sometimes flags bogus issues.

11. Click the Advanced button.

12. Click the Legacy radio button.

This causes the compiled XOP to be stored where we expect it, in the WaveAccess/Xcode6/Debug or WaveAccess/Xcode6/Release folder.

13. Click Done to dismiss the Advanced Project Settings dialog.

14. Click Done to dismiss the Project Settings dialog.

15. Click the project icon.

This is the WaveAccess icon at the top of the project navigator pane on the left side of the Xcode main window.

The project editor pane listing PROJECT and TARGET appears if it was not already visible.

If you don’t see PROJECT and TARGET, click the small rectangular button as explained under **Xcode 6 Project Build Settings** on page 37.

16. Click the disclosure triangles under the WaveAccess project icon to reveal the files that comprise the project.

17. Click the WaveAccess icon under PROJECT to display the project build settings.

18. Click the Build Settings tab to display the project build settings.

19. Click the All and Combined buttons.

20. Under Architectures, choose 32-Bit Intel.

Xcode 6 does not compile for PowerPC. The compiled XOP will run on Intel Macintoshes only.

21. Under Base SDK, choose “Latest OS X”.

22. Under Compiler for C/C++ choose “Default Compiler”.

23. In the Deployment section, Under OS X Deployment Target choose 10.6.

The XOP will run under OS X 10.6 or later.

24. Click the WaveAccess icon under TARGET to display the target settings.

25. Click the Build Settings tab.

26. Click the All and Combined buttons.

27. For Architectures, choose “32-Bit Intel” if it is not already selected.

28. For Base SDK, choose “Latest OS X” if it is not already selected.

29. In the Build Options section, for Compiler for C/C++ choose “Default compiler” if it is not already selected.

30. Find the Other Warning Flags setting in the Apple LLVM Custom Compiler Flags section and add:
-Wno-parentheses

31. Choose Product→Build.

This is the same as choosing Product→Build For→Build For Running.

The WaveAccess XOP now compiles in Xcode 6.

XOPs in Visual C++ 2008 through 2013

This version of the manual describes Windows XOP development with Visual C++ 2015 only. For details on using earlier versions of Visual C++, download the XOP Toolkit 6.30 manual from:

<ftp://ftp.wavemetrics.net/IgorXOPToolkit/Manual/Old/XOPToolkitManual630.pdf>

XOPs in Visual C++ 2015

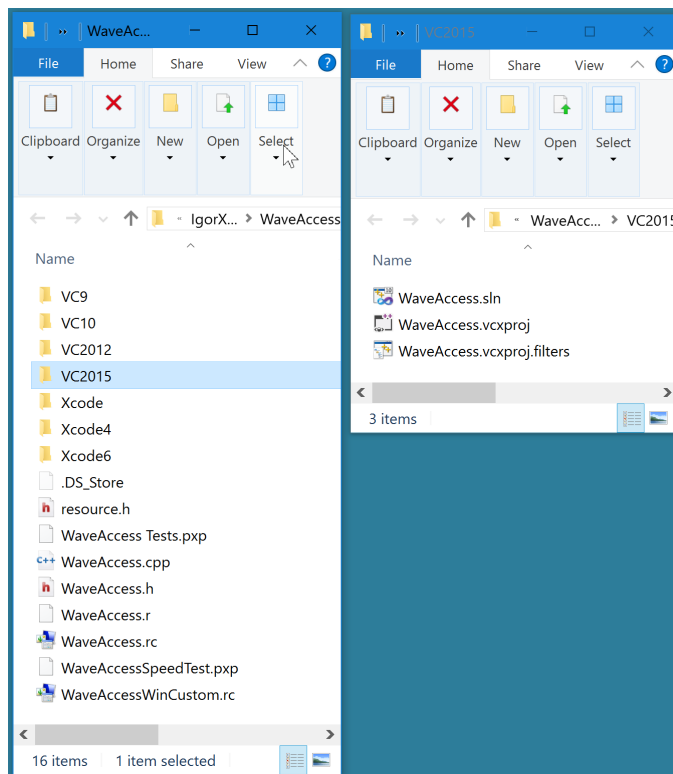
Microsoft Visual C++ is a C and C++ development system that can build programs for Microsoft Windows. It runs on Windows and produces Windows XOPs.

The express (free) edition of Visual C++ is fine for most 32-bit and 64-bit XOP development.

XOP Projects in Visual C++ 2015

This section provides the background information needed to understand how to create XOP projects in Visual C++. For step-by-step instructions on creating a project, see **Creating a New Project** on page 21.

We will use the WaveAccess sample XOP as a case in point. The WaveAccess folder is inside the IgorXOPs6 folder and looks like this:



The Visual C++ project files are inside the VC2015 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/VC2015 folder.

Chapter 3 — Development Systems

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

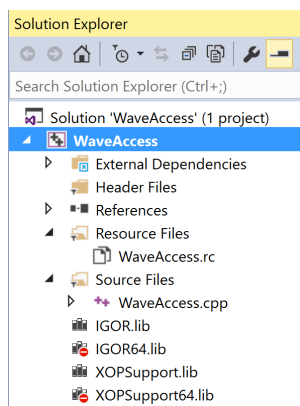
WaveAccess.cpp contains the project source code while WaveAccess.h contains the project headers.

WaveAccess.rc is the main Windows resource file. It #includes resource.h (created by Visual C++), and WaveAccessWinCustom.rc (created by the XOP programmer). WaveAccess.rc contains standard Windows resources such as the version resource and menu and dialog resources, if any. 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. The use of these resource files is further explained under **XOP Resources** on page 51.

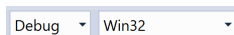
WaveAccess.r contains the Macintosh resources and is not used on Windows.

The .DS_Store file is a Macintosh desktop file and is not needed on Windows. It is present because the XOP Toolkit is built on a Macintosh volume.

This screen shot of the Visual C++ 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 as a 32-bit or 64-bit XOP. You select which will be built by choosing Build → Configuration Manager or using the configuration popup menus in the Visual C++ toolbar:



Both the debug and release 32-bit configurations create an XOP in the VC2015 folder 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 should build the release configuration which may run significantly faster.

To compile the 64-bit version, choose x64 instead of Win32 from the platform popup menu and then compile the project. This creates a 64-bit version (e.g., WaveAccess64.xop) in the VC2015 folder. 64-bit XOPs are discussed in detail under **64-bit XOPs** on page 181.

Visual C++ 2015 Project Settings

If you choose View → Solution Explorer and then right-click the WaveAccess icon and then choose Properties, Visual C++ displays the project settings. Here is a discussion of the significant settings which need to be changed from the default using WaveAccess as an example.

If you change a setting, you should change it for both the debug and release configurations and for the 32-bit and 64-bit platforms unless otherwise noted.

General Properties

Output Directory: .\

This tells Visual C++ to put the compiled XOP in the VC2015 folder.

Target Name: WaveAccess for 32-bits; WaveAccess64 for 64-bits

Target Extension: .xop

Character Set: Use Multi-Byte Character Set

Debugging Properties

Command: Enter the path to your Igor.exe application file for the 32-bit platform, Igor64.exe for the 64 bit platform.

Visual C++ will launch Igor when you ask to debug your XOP.

C/C++ Properties / General Category

Additional Include Directories: Add ..\..\XOPSupport

This allows #include statements to find the XOPSupport header files. The path is relative to the folder containing the .vcxproj project file.

The final setting will contain “..\..\XOPSupport;%(AdditionalIncludeDirectories)”. The latter part is automatically added by Visual C++ by default.

C/C++ Properties / Preprocessor

Preprocessor Definitions: Add _CRT_SECURE_NO_WARNINGS

This setting must be added separately for the debug and release configurations because other preprocessor definitions are configuration-specific.

C/C++ Properties / Code Generation Category

Runtime Library: Multi-Threaded (/MTd) for the debug configuration; Multi-Threaded (/MT) for the release configuration.

Do not use a DLL version of the runtime library as the DLL is not present on all systems.

An exception to this rule is that, if you have to link with a library compiled with a different version of Visual C++, you may get link errors relating to missing C runtime routines. In this case you have no choice but to use the DLL versions of the C runtime. Then anyone using your XOP will need to have the C runtime DLLs for your Visual C++ version. For further information, search the web for “Visual C++ redistributable”.

Struct Member Alignment: Default

See **Structure Alignment** on page 209 for details.

C/C++ Properties / Advanced

Disable Specific Warnings: Add “4312;” without the quotes.

This disables “type casting from ‘a’ to ‘b’ of greater size” warnings.

Linker Properties / General Category

Output File: WaveAccess.xop for 32-bits; WaveAccess64.xop for 64-bits

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\VC2015). 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, regardless of configuration.

Linker Properties / Input Category

Additional Dependencies: Add version.lib.

The final setting will contain “version.lib;%(AdditionalDependencies)”. The latter part is automatically added by Visual C++ by default.

Linker Properties / System Category

Enable Large Addresses: Yes (/LARGEADDRESSAWARE)

This allows the XOP to address up to 4 GB when running with the 32-bit version of Igor instead of the default 2 GB.

Linker Properties / Advanced Category

Target Machine: MachineX86 for 32-bit XOPs; MachineX64 for 64-bit XOPs.

Resources Properties / General Category

Additional Include Directories: ..\..\XOPSupport

This allows #include statements to find the XOPResources.h file. The path is relative to the folder containing the .vcxproj project file.

Visual C++ 2015 XOPSupport Warnings

When you build an XOP in Visual C++, you may see LNK4099 warnings saying that XOPSupport.lib is missing debugging information. This warning is generated when you link the debug configuration of your XOP to the XOPSupport library if the library’s project database file, named vc90.pdb, is not available.

There is no way to suppress the LNK4099 warning.

The warning is of no consequence unless you want to step into XOPSupport routines while debugging. In that case, you can fix the problem by opening IgorXOPs6\XOPSupport\VC2015\XOPSupport.sln in Visual C++ and rebuilding the 32-bit and 64-bit XOPSupport libraries using the Debug configuration.

Debugging a Visual C++ 2015 XOP

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

1. Open the solution (e.g., WaveAccess.sln) in Visual C++.
2. Use Build→Configuration Manager or the configuration popup menus to activate the debug configuration of the project.
3. In the Solution Explorer window, right-click the WaveAccess icon, choose Properties, click the Debugging category, and set the Command setting for the debug configuration to the path to your Igor.exe or Igor64.exe file.
4. Build the XOP by choosing Build→Build Solution.
5. On the desktop, create a shortcut from the executable XOP file (e.g., WaveAccess\VC2015\WaveAccess.xop or WaveAccess64.xop) that you just built and put the shortcut in the “Igor Extensions” or “Igor Extensions (64-bit)” folder.
6. Back in Visual C++, open your main source file (e.g., WaveAccess.cpp), and set a breakpoint at the start of the XOPMain function.

7. Press F5. The Visual C++ debugger will launch Igor Pro.
8. 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 XOPMain function where you set the breakpoint.
9. 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.

Converting an Old Visual C++ Project

Microsoft made a change in the Visual C++ 2015 compiler that breaks existing XOPs. Many XOPs use the CR_STR macro provided by the XOPSupport header files. In Visual C++ 2015, you need to have a space before the CR_STR macro. For details, see **Visual C++ 2015 Projects** on page 487.

For other conversion instructions, see **Project Changes For XOP Toolkit 6** on page 457.

Igor/XOP Interactions

Overview

The code on page 50 is a simplified sketch of 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. Read the code now but don't worry about the details which we will explain next.

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 identified by their ".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 XOPMain function, passing it an IORecHandle. The XOPMain function does all of the XOP's initialization including calling the following XOPSupport routines.

Routine	Description
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 subsequent calls to the XOP except for external operation and external function invocations.
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.

Chapter 4 — Igor/XOP Interactions

```
// 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.cpp for Macintosh and Windows.

extern "C" 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, . . ., (void*)ExecuteXOP1, . . .)
}

extern "C" void
XOPEntry(void)
{
    XOPIORecResult result = 0;
    switch (GetXOPMessage()) {
        // Handle message
    }
    SetXOPResult(result);
}

HOST_IMPORT int
XOPMain(IORecHandle ioRecHandle) // The main function
{
    XOPInit(ioRecHandle);
    SetXOPEntry(XOPEntry);

    if (igorVersion < 620) {        // Requires Igor Pro 6.20 or later.
        SetXOPResult(IGOR_OBSOLETE);
        return EXIT_FAILURE;
    }

    RegisterXOP1();
    SetXOPResult(0L);
    return EXIT_SUCCESS;
}
```


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. Chapter 5 and Chapter 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.

In the example above, the XOPEntry and ExecuteXOP1 functions were declared `extern "C"`. This is appropriate because these functions are called from C code in Igor using C calling conventions. If you are coding the XOP in C instead of C++, use `static` in place of `extern "C"`.

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 12
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. These resource types are used on both Macintosh and Windows.

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.

Simple XOPs typically have an XOPI resource, an XOPC or XOPF resource a STR#,1100 resource and possibly a STR#,1101 resource.

Creating Resources on Macintosh

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

In days gone by you could graphically edit resources using Apple's ResEdit program. With Mac OS X, Apple moved away from using resources to describe user interface items like menus, controls, dialogs and windows. Consequently Apple abandoned ResEdit. To replace resources, Apple 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 6 and Igor Pro 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. Xcode includes a resource compiler that compiles any .r files in the project. The resulting resources are stored in the .rsrc file in the XOP's package folder.

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

Creating Resources on Windows

On Windows, Visual C++ creates resources by compiling resource (.rc) files. You can create a resource file either by entering text directly into the file or by using the development system's graphical resource editor. However the express editions of Visual C++ do not support graphical resource editing so you must enter text directly into the file using the source code editor.

The graphical resource editor allows you to graphically edit standard Windows resources such as icons, menus and dialogs. The source code editor allows you to edit a resource file as plain text, which is better for editing text-oriented resources such as XOP function declarations, operation declarations and error strings.

To open a resource file for graphical editing, double-click its icon in the Solution Explorer. To open it for text editing, right-click it and choose View Code.

On Windows, the XOP-specific resources are stored in a separate resource file which is edited as a text file. For example, the XOPI sample XOP has two resource files, an XOPI.rc file that stores standard Windows resources (e.g., version and menu resources) and an XOPIWinCustom.rc file that stores WaveMetrics-defined resources (e.g., XOPI, XOPC and XOPF resources).

Visual C++ allows you to add only one resource file to your project. Consequently, the main resource file (XOPI.rc) contains an include statement which includes the XOP-specific resource file (XOPIWinCustom.rc). You can see this by opening the XOPI.rc file using the source code editor. You will see this line:

```
#include "XOPIWinCustom.rc"
```

Visual C++ generates a file named resource.h, which is also included by the main resource 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.
resource XOPI (1100) { // 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,                    // Obsolete - set to zero.
    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 66 for details.

The next field tells Igor what development system was used to compile the XOP, which previously affected how Igor called the XOP. It is currently not used but must be present.

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.

How Igor Calls an XOP

Igor calls your XOP by calling one of four routines:

Event	How XOP Is Called
INIT message	Igor sends the INIT message by calling your <code>XOPMain</code> function.
Call external operation	To invoke an external operation added by your XOP, Igor calls a function that you designate directly at initialization time as explained in Chapter 5, Adding Operations .
Call external function	To invoke an external function added by your XOP, Igor calls a function that you designate directly at initialization time as explained in Chapter 6, Adding Functions .
All other messages	Igor sends all other messages by calling your <code>XOPEntry</code> routine.

Calling external operations and functions is explained in chapters 5 and 6. The rest of this chapter explains messages sent to your XOP by calling your `XOPMain` function or your `XOPEntry` routine.

Basic XOP Messages

Here are the basic messages that Igor can send to your XOP's XOPEntry routine, or your XOPMain function in the case of the INIT message. The constants (INIT, IDLE, CMD, etc.) are defined in XOP.h. When Igor calls your XOPEntry routine, you call the **GetXOPMessage** XOPSupport routine to determine what message is being sent. 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 237.

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 except for INIT.

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 code. A result code of 0 indicate success while a non-zero result code indicates an error. See **XOP Errors** on page 57 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	<p>Tells your XOP that it needs to initialize itself.</p> <p>Arguments: None.</p> <p>Result: Error code from your initialization.</p> <p>This is the first message received by every XOP when it is loaded. This message is sent by calling your XOPMain function. Subsequent messages are sent by calling your XOPEntry routine.</p> <p>If your XOP returns a non-zero result via the SetXOPResult XOPSupport routine, Igor will unload your XOP immediately. Therefore you should do any necessary cleanup before returning a non-zero result.</p>
IDLE	<p>Tells your XOP to do its idle processing.</p> <p>Arguments: None.</p> <p>Result: None.</p> <p>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.</p> <p>See Handling Recursion on page 64 for a discussion of recursion issues involving IDLE messages.</p>
MENUITEM	<p>Tells your XOP that its menu item has been selected.</p> <p>Argument 0: Menu ID of menu user selected.</p> <p>Argument 1: Item number of item user selected.</p> <p>Result: Your result code from operation.</p> <p>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.</p> <p>See Chapter 8, Adding Menus and Menu Items, for details on how to respond to this message.</p>
MENUENABLE	<p>Tells your XOP to enable or disable its menu items.</p>

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.

An XOP receives the MENUENABLE message if its window is the front window even if the XOP adds no menu items to Igor. This allows the XOP to enable or disable Igor menu items such as items in the Edit or File 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 created 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.

The OBJINUSE message is sent only from the main Igor thread. A complementary technique is available for preventing the deletion of waves used in an Igor preemptive thread. See **Wave Reference Counting** on page 128 for details.

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.

See **Preventing the Killing of a Wave** on page 128 for further discussion of the OBJINUSE message.

FUNCADDRS

Asks your XOP for the address of one of its external functions.

Argument 0: Function index number starting from 0.

Result: Address of function or NULL.

See Chapter 6, **Adding Functions**, for details.

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	<p>Tells your XOP that the user selected New Experiment from the File menu.</p> <p>Arguments: None.</p> <p>Result: None.</p> <p>When the user creates a new experiment your XOP should set its settings to their default state.</p>
MODIFIED	<p>Asks your XOP if it has been modified since the last NEW, LOAD or SAVE message and needs to save settings or documents.</p> <p>Arguments: None.</p> <p>Result: 0 if your XOP is not modified, 1 if modified.</p> <p>You need to respond to this message if your XOP saves settings or documents as part of an experiment. Otherwise, ignore it.</p>
CLEAR_MODIFIED	<p>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.</p> <p>Arguments: None.</p> <p>Result: None.</p>
SAVE	<p>This message is obsolete. Igor Pro will not send it.</p>
LOAD	<p>This message is obsolete. Igor Pro will not send it.</p>
SAVESETTINGS	<p>Allows your XOP to save settings as part of an experiment.</p> <p>Argument 0: Experiment type.</p> <p>Argument 1: Save type.</p> <p>Result: Handle to your XOP's settings or NULL.</p> <p>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 61 for details.</p> <p>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.</p> <p>The experiment type will one of the following which are defined in IgorXOP.h:</p> <p>EXP_PACKED</p> <p>EXP_UNPACKED</p> <p>The save type will be one of the following which are defined in IgorXOP.h:</p> <p>SAVE_TYPE_SAVE</p> <p>SAVE_TYPE_SAVEAS</p>

	SAVE_TYPE_SAVEACOPY SAVE_TYPE_STATIONERY
LOADSETTINGS	<p>Allows your XOP to load settings it saved in an experiment.</p> <p>Argument 0: Handle to data saved in experiment file or NULL. Argument 1: Experiment type. Argument 2: Load type. Result: None.</p> <p>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 61 for details.</p> <p>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.</p> <p>The experiment type will one of the following which are defined in IgorXOP.h:</p> <p>EXP_PACKED EXP_UNPACKED</p> <p>The load type will be one of the following which are defined in IgorXOP.h:</p> <p>LOAD_TYPE_OPEN LOAD_TYPE_REVERT LOAD_TYPE_STATIONERY LOAD_TYPE_MERGE</p>

XOP Errors

The XOP protocol facilitates providing helpful error messages when something goes wrong in your XOP. You return an error code to Igor in response to the INIT, CMD, FUNCTION and MENUITEM messages by calling the SetXOPResult XOPSupport routine. Your direct external operations and direct external functions return error codes through the return statement of those routines. If you return a non-zero value, Igor displays an error dialog. Igor displays the dialog when your XOP returns, not immediately.

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 60 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:

Type	Range	Description
Igor error code	-1 to 9999	Error codes used by Igor itself. Defined in IgorErrors.h.

Chapter 4 — Igor/XOP Interactions

Type	Range	Description
Mac OS error codes	-32768 to -2	Defined by Macintosh Carbon API but used on Macintosh and Windows.
XOP error codes	10000 to 10999	Defined by the XOP programmer.

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 `IgorErrors.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.

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

XOP custom 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 12. 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 under **Adding Custom Errors** on page 60.

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 return 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 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 you use `WMGetLastError` instead of `GetLastError`, you 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`, you 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 direct external functions must return an error code as their 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 OLD_IGOR 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 6.20 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 6.20 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 external operation or direct external 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.

Eliminate Empty XOP Error Strings

In Igor Pro 6.20 the internal method for handling XOP error messages was changed to support external operations and functions running in threads. In 6.20 or later, Igor copies all XOP error strings from the XOP's STR# 1100 resource into Igor memory at launch time.

If you have an empty error string in your STR# 1100 resource, this will cause Igor to ignore any error strings after the empty one. If you have an unused error message that is currently set to "" in your STR# 1100 resource, change it to something like "-unused-".

Also make sure that your XOP error codes, defined in your main .h file, are contiguous starting from (FIRST_XOP_ERR + 1).

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 XOP 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 the 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. Also, it is a good idea to make the first field of your structure a version field.

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 state. When the user opens an experiment, Igor sends the LOADSETTINGS message. When the user saves an experiment, Igor sends the SAVESETTINGS message.

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.cpp 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 create 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 the handle is NULL, set your XOP to its default state. 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, PowerPC 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 209.

If you are loading settings stored on a PowerPC while running on an Intel machine or vice versa, 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. Also, it is a good idea to make the first field of your structure a version field.

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 XOPMain function calls XOPInit.

You should never deal with the IORecHandle directly since the routines in XOPSupport.c do this for you.

Resident and Transient XOPs

The `XOPType` field of the `IORecHandle` indicates to Igor the capabilities or mode of the XOP. You can set this field using the **SetXOPType** XOPSupport routine but this is not necessary for most XOPs. 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 initialized by Igor to `RESIDENT`. This means that the XOP will stay in memory indefinitely. All XOPs that add operations or 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 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.

See **Handling Recursion** on page 64 for a discussion of recursion issues involving IDLE messages.

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 `XOPMain` 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 `XOPMain` 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.

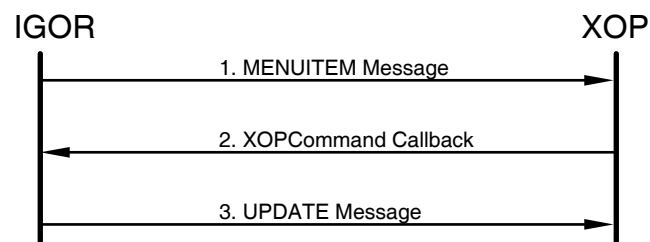
Some messages from Igor require that you return a result. Depending on the particular message, the result may be an error code, a pointer or a handle. You return the result by calling `SetXOPResult`. This XOPSupport routine stores the result in the result field of the `IORecHandle`.

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 and similar routines. These routines 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.

For example, if your XOP does `IDLE` processing, it is possible to receive a second `IDLE` message from Igor while you are processing the first. One way for this to happen is if, as part of your `IDLE` processing, you use **XOPCommand** to call a user-defined function that calls Igor's `DoXOPIdle` operation. `DoXOPIdle` sends `IDLE` messages to any XOP that has requested to receive them. There may be other circumstances that lead to Igor sending you an `IDLE` message while you are processing an earlier `IDLE` message.

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 for example, 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 **GetXOPItem** 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.

Similarly, to prevent recursion involving `IDLE` messages, you would set a global when you receive an `IDLE` message and clear it when you finish `IDLE` processing. If you receive another `IDLE` message while the global is set, you ignore the subsequent `IDLE` message.

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 passed as function parameters
- Via pointers to local data (on the stack)
- Via pointers to data in the heap
- Via handles to data in the heap

Chapter 4 — Igor/XOP Interactions

Values (e.g., ints, doubles) and pointers are built-in elements of the C programming language. They are documented in C programming books.

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. On Windows 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 197.

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 XOPMain function, sets a global variable, `igorVersion`, to the version number of the currently running Igor. For version 6.20, `igorVersion` will be 620.

All XOPs compiled by XOP Toolkit 6.4 require Igor Pro 6.20 or later. Consequently the XOPMain function in each XOP should contain code like this:

```
if (igorVersion < 620) {           // Requires Igor Pro 6.20 or later.
    SetXOPResult(IGOR_OBSOLETE);
    return;
}
```

This test must be placed after your call to **XOPInit** and **SetXOPEntry**.

For example, if your XOP uses features added to Igor in version 6.30, you must restrict your XOP to running with Igor Pro 6.30 or later. Put the following in your XOP's XOPMain function:

```
if (igorVersion < 630) {           // Requires Igor Pro 6.30 or later.
    SetXOPResult(IGOR_OBSOLETE);
    return;
}
```


IGOR_OBSOLETE is an Igor error code, defined in IgorErrors.h. Your XOP's help file should specify what version of Igor your XOP requires.

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 does not load the XOP. This provides a way for Igor to make sure that your XOP is sufficiently compatible that it is OK to call its XOPMain 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 6 work with XOPs whose XOP protocol version is 2, 3 or 4. The current value of XOP_VERSION is 4.

Adding Operations

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.

Flags can consist of one to four letters (/A, /AB, /ABC or /ABCD).

<parameter list> consists of one or more 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 extension=".dat" dayOfWeek=Monday size={3.4,5.2}

A parameter must be separated from the next with a comma.

Chapter 5 — Adding Operations

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

Direct Method Versus CMD Message Method

Igor Pro 5 added a feature called “Operation Handler” which greatly simplified the process of creating an external operation. In addition, Operation Handler made it possible to call external operations directly from Igor user-defined functions. Prior to Igor Pro 5, to call an external operation from a user-defined function you had to use a kludge involving Igor’s Execute operation.

Using Operation Handler, you specify an ExecuteOperation function that Igor calls directly. This is called the direct method. Prior to Operation Handler, you added an external operation to Igor by responding to the CMD message. This is called the CMD message method.

The CMD message method is still supported by Igor for the benefit of very old XOPs but is not otherwise used and is not documented in this version of the XOP Toolkit manual.

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, specifies the names of the operations that the XOP adds to Igor.

If the user invokes one of these operations from Igor’s command line or from a procedure, Igor loads the XOP into memory and calls its XOPMain function (if it’s not already loaded). The XOP’s XOPMain function calls **RegisterOperation**, an XOPSupport routine through which the XOP tells Igor the syntax of the operation and the address of the XOP function, called the “ExecuteOperation function”, which Igor should call to implement it.

Next Igor parses the operation parameters and uses them to fill a structure called the “runtime parameter structure”. It then calls the XOP’s ExecuteOperation 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) {
    {
        "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 CMD message method documented in the XOP Toolkit 3.1 manual. Since the CMD message method is obsolete, all new operations use Operation Handler and therefore the compilableOp flag must be present in the operation category specifier for each operation.

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
statsOp	16384	Statistics
threadSafeOp	32768	Thread-safe operation

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-defined function.

Command line and macro operations are interpreted while user-defined function operations are compiled and then later executed. 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-defined 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.

Operation Handler handles all of the details of compiling your operation in a user-defined 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, generates the starter code and stores it in the clipboard. Anything previously in the clipboard is overwritten. You can then paste the starter code 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=6 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 XOPMain function

Before reading further, 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.

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.

<u>SampleOp /F=3 /K={A, 3.2}</u> flags	<u>wave0, wave1</u> simple main parameters
<u>SampleOp /F=3 /K={A, 3.2}</u> flags	<u>waves={wave0, wave1}</u> 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”.

Igor Pro 6.30 and later support “qualified” keywords. For example:

```
ModifyGraph mode = 3 // Unqualified: Set all traces to markers mode
ModifyGraph mode(wave0) = 3 // Qualified: Set wave0 only to markers mode
```

Chapter 5 — Adding Operations

Here `wave0` is the "qualifier". It narrows the scope of the mode keyword. Qualified keywords are discussed in detail under **Qualified Keywords** on page 80.

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.

Parameters can be required or optional. Optional parameters are described under **Optional Parameters** on page 75. Required parameters must be supplied when the operation is compiled or interpreted. If required parameters are missing, Operation Handler 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:pName /Y={number:n1,number:n2} key1={number:n3,string:s1}
```

This says that the operation name is `SampleOp`, and 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:pName                    // A flag group with one parameter
/Y={number:n1,number:n2}        // A flag group with two parameters
keyword1={number:n3,string:s1}  // A keyword group with two parameters
```

The `/P` flag takes one parameter of type `name`. The parameter's mnemonic name is `pName`. The mnemonic name is used as a field name in the runtime parameter structure that is passed to your `ExecuteOperation` when your operation is invoked.

Similarly, the `/Y` flag takes two parameters of type `number` with mnemonic names `n1` and `n2`. The `keyword1` keyword takes two parameters, a number whose mnemonic name is `n3` and a string whose mnemonic name is `s1`. In real life you would choose more meaningful mnemonic names.

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 "suffix" 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:n1 /B=string:s1
```

In main parameters, one group is separated from the next by a comma:

```
SampleOp number:n1, string:s1
SampleOp key1=number:n1, key2=string:s1
```

There are two exceptions to this. The keywords `'as'` and `'vs'` can be used in place of comma between one simple main parameter and the next. These are special cases that supports syntax like this:


```
Save wave:w1 as string:s1
```

```
Display wave:yWave vs wave:xWave
```

These special cases do not work if the first parameter is numeric.

The 'as' or 'vs' keyword must appear both in the command template and in the actual command.

The 'vs' keyword was added in Igor Pro 6.30. If you use this feature, you must check Igor's version as described under **Checking Igor's Version** on page 66.

The recognized parameter type keywords are:

```
number
string
name
doubleName          // Requires Igor Pro 6.00 or later
wave
waveRange
varName
dataFolderRef        // Requires Igor Pro 6.20 or later
dataFolderAndName
structure            // Requires Igor Pro 5.03 or later
```

Most operations will use number, string, name and wave parameters. The other types are less common.

The name parameter type is used to get the name of an object that is not part of Igor's data hierarchy, such as a symbolic path name or a picture name.

The doubleName parameter type is used to allow the user to specify a compound function name that includes a module name (e.g., MyModule#MyFunction) or a compound picture name (e.g., Proc-Global#MyPicture).

The wave type is used to get a reference (handle) to an existing wave.

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. VarName should not be used for normal input parameters – use number instead.

The dataFolderRef type is used to get a reference (handle) to an existing data folder. Support for dataFolderRef parameters was added in Igor Pro 6.20. If you use this feature, you must check Igor's version as described under **Checking Igor's Version** on page 66.

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:

Chapter 5 — Adding Operations

```
// Optional flag or keyword parameters
SampleOp /A[=<parameters>]
SampleOp key1[=<parameters>]
SampleOp key1[( <parameters> )]=<parameters>

// Normal optional parameters
SampleOp /A={number:n1 [,string:s1, wave:w1]}
SampleOp key1={number:n1 [,string:s1, wave:w1]}
SampleOp number:n1 [,string:s1, wave:w1]

// Array-style optional parameters
SampleOp /A={number:n1, string[3]:s1}
SampleOp key1={number:n1, string[3]:s1}
SampleOp number:n1, string[3]:s1
```

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. With a qualified keyword, it also allows the user to supply or omit the qualifier.

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]:waves as string
```

In this case at least one wave must be specified in the command before the 'as' keyword.

Similarly when creating a Display type of operation, you can combine the 'vs' keyword and the array-style optional parameters like this:

```
Display wave[100]:waves as wave:xWave // vs requires Igor Pro 6.30 or later
```

In this case at least one wave must be specified in the command before the 'vs' keyword.

The 'vs' keyword was added in Igor Pro 6.30. If you use this feature, you must check Igor's version as described under **Checking Igor's Version** on page 66.

Other than the usages shown above, no other use of brackets to indicate optional parameters is allowed. For example, the brackets here are not needed or allowed:

```
SampleOp [ /A/B=number:n1/C=string:s1 ] // 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 an optional 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 runtime, you test to see if the second parameter was set.

Mnemonic Names

The mnemonic names that you provide in the template are used to automatically generate the source code for your runtime parameter structure. See **Creating Starter Code** on page 72. Each mnemonic name becomes the name of a field in the structure. Therefore mnemonic names in a given template must be distinct.

The Runtime Parameter Structure

When you create your starter code, Operation Handler defines a structure into which it will store parameters and other values at runtime. When your operation is invoked, Igor passes 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:key1StrH},
        key2={name:key2Name,wave:key2WaveH}
```

To generate starter code you would pass this template to the ParseOperationTemplate operation:

```
ParseOperationTemplate /C=6 /S=1 /T "<template here>"
```

This stores starter code in the clipboard. The runtime parameter structure part of the starter code looks like this:

```
#pragma pack(2)                                // Igor structures are 2-byte-aligned.

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;
```

Chapter 5 — Adding Operations

```
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-defined function.
int calledFromMacro;       // 1 if called from a macro.
};
typedef struct SampleOpRuntimeParams SampleOpRuntimeParams;
typedef struct SampleOpRuntimeParams* SampleOpRuntimeParamsPtr;

#pragma pack()                // Reset structure alignment.
```

It is critical that the structure's fields 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. Therefore it is best to use `ParseOperationTemplate` to regenerate the code for the structure after changing the command template as this guarantees that the template and structure will be consistent.

All structures passed between Igor and an XOP use two-byte packing. You must use the pragmas 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 parameter there is a corresponding field of the appropriate type in the runtime parameter structure:

Parameter Type	Field Type
number	double
string	Handle
name	char[MAX_OBJ_NAME+1]
doubleName	char[MAX_LONG_NAME+1]
wave	waveHndl
waveRange	WaveRange structure
varName	char[MAX_OBJ_NAME+1]
dataFolderRef	DataFolderHandle
dataFolderAndName	DataFolderAndName structure
structure	pointer to structure

NOTE: At runtime, the handles corresponding to `string`, `wave` and `dataFolderRef` parameters can be `NULL` as can the pointer corresponding to a structure parameter. This can happen if the caller passes an uninitialized reference or a reference whose initialization failed to you. It can also

happen if the caller passes * for the parameter which signifies “default”. Your code must test for NULL parameters as described in the following sections.

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. For example, if the command template includes

```
/A
```

this is represented in the runtime parameter structure as:

```
// Parameters for /A flag group.
int AFlagEncountered;
```

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 runtime, 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 flag parameter group can be set only once by a single command. Operation Handler returns an error if a command tries to use a given flag group twice.

Igor Pro 6.30 and later allow a command to use the same keyword multiple times if all keyword parameters are array-style parameters. Previous versions of Igor treated this as an error. If all keyword parameters are array-style parameters with n elements, a single command can use the keyword up to n times. Operation Handler generates an error if the keyword is used more than n times.

Chapter 5 — Adding Operations

For keyword parameter groups, if the same keyword is used more than once, the Encountered field will reflect the first occurrence of the keyword and there will be gaps in the sequence.

Qualified Keywords

Igor Pro 6.30 and later support "qualified" keywords. For example:

```
ModifyGraph mode = 3           // Unqualified: Set all traces to markers mode
ModifyGraph mode(wave0) = 3    // Qualified: Set wave0 only to markers mode
```

Here wave0 is the "qualifier". It narrows the scope of the mode keyword.

If you use this feature, you must check Igor's version as described under **Checking Igor's Version** on page 66.

The qualifier can consist of any number of parameters of any type. It appears in parentheses immediately after the keyword. You can not use brackets or braces to introduce the qualifier.

The entire qualifier can be optional. If the qualifier consists of more than one parameter, all parameters after the first can be optional.

In the general case, the template for a qualified keyword looks like this:

```
SampleOp keyword [(<qualifier1>[,<qualifier2>]) ] [= {<value1>[,<value2>]}]
```

Any pair of brackets can be omitted in which case the corresponding element will be required rather than optional.

The **first** left bracket and its matching right bracket make the entire qualifier optional.

The **second** left bracket and its matching right bracket make the second parameter of the qualifier optional.

The **third** left bracket and its matching right bracket make the entire righthand side optional.

The **fourth** left bracket and its matching right bracket make the second righthand parameter optional.

You can make the entire qualifier optional but the first parameter of the qualifier can not be optional:

```
// OK - Entire qualifier optional
SampleOp key1 [(number:num1,number:num2)]

// OK - Second qualifier parameter optional
SampleOp key1 [(number:num1[,number:num2])]

// Bad - First qualifier parameter can not be optional
SampleOp key1 [( [number:num1,number:num2] )]
```

In addition to normal optional parameters, you can use array-style optional parameters:

```
ModifyGraph mode [(name[10]:traceName)] = {number[10]:modeCode}
```

If the qualifier is required and the only qualifier parameter is an array-style optional parameter, as in this template:

```
SampleOp key1 (number[3]:array)           // Array-style optional parameter
```

then you can invoke the operation with no parameters:

```
SampleOp key1                               // Bad - qualifier required
SampleOp key1()                             // OK - no parameters
SampleOp key1(1)                           // OK - one parameter
SampleOp key1(1,2)                         // OK - two parameters
SampleOp key1(1,2,3)                       // OK - three parameters
```

When all keyword parameters are array-style optional parameters, there are two ways to specify the qualifier when invoking the operation. Using the `ModifyGraph` example:

```
ModifyGraph mode(wave0,wave1) = {3,4}      // Set mode for two traces
ModifyGraph mode(wave0)=3, mode(wave1)=4    // Set mode for two traces
```

Both commands return the following via the runtime parameter structure:

```
traceName[0]: wave0
traceName[1]: wave1
traceNameParamsSet[0]=1, traceNameParamsSet[1]=1
modeCode[0]: 3
modeCode[1]: 4
modeCodeParamsSet[0]=1, modeCodeParamsSet[1]=1
```

All other elements of `traceNameParamSet` and `modeCodeParamSet` will be 0.

It is permitted to use the same keyword twice in a single command only if all keyword parameters are array-style parameters.

The user may invoke the operation like this:

```
ModifyGraph mode(wave0,wave1) = 3
```

In this case, two elements of the `traceName` array in the runtime parameter structure will be set but just one element of the `modeCode` array will be set. You can choose to treat this as if the user executed:

```
ModifyGraph mode(wave0)=3, mode(wave1)=3
```

or you can consider this an error and return an error code. Your code should not assume that the same number of parameters were passed for both array-style parameters.

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 the contents of a global string variable to you and the global string variable 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 parameter 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. See **Understand the Difference Between a String in a Handle and a C String** on page 237 for further discussion.

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 cor-

responding 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.

Double Name Parameters

Double names are rarely used so most XOPs will not need to use this type of parameter.

Igor double names are used for combining a module name and a function or proc pict name in one compound name, for example, MyModule#MyProcPict. They consist of up to MAX_LONG_NAME characters. The user can specify a double 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.

When running in an independent module, if a simple name is specified for a doubleName parameter, Igor automatically prepends the independent module name. For example, if a function is running in an independent module named MyIndependentModule and it executes:

```
SampleOp functionName=MyFunction
```

you receive "MyIndependentModule#MyFunction" as the parameter.

Wave Parameters

Waves are passed to you as wave handles of type waveHndl. Wave handles always belong to Igor. You must never dispose or directly modify a wave handle. You operate on a wave by passing its handle to an XOPSupport routine.

The handle for a wave parameter can be NULL. This would happen, for example, if the user passed an uninitialized wave reference (WAVE) or a wave reference whose initialization failed to your operation. 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 NULL_WAVE_OP 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. See the **WaveType** and **MDGetWaveDimensions** functions.

Wave Range Parameters

Wave range parameters allow the user to specify a range of a 1D wave. They 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 NULL_WAVE_OP, unless you want to allow a null wave. As with the wave parameter, the user can use * in place of the wave name in which case the waveH field will be NULL. Also as with the wave parameter, you must check the wave's type to make sure it is a type you can handle. See **WaveType**.

waveH is treated as 1D regardless of its actual dimensionality.

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.

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`.

```

IndexInt 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. For example, an operation that opens a file can return a file reference number via a `varName` parameter, as Igor's `Open` operation does.

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-defined function, the name of an NVAR or SVAR.

When called from the command line or from a macro, the `varName` field in the runtime parameter structure contains an actual variable name. When called from a user-defined 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 XOPSupport routines designed to handle it as described in this section.

At runtime you can determine the type of the variable by calling **VarNameToDataType** which returns numeric (NT_FP64), complex numeric (NT_FP64 | NT_CMPLX) or text (0).

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.

Chapter 5 — Adding Operations

In Igor Pro 6.10, the ability to retrieve values from numeric and string variables was added through the **FetchNumericDataUsingVarName** and **FetchStringDataUsingVarName** XOPSupport functions. These can be used to implement external operation parameters that function both as inputs and outputs. For example, you would use **FetchNumericDataUsingVarName** to retrieve the initial value of a numeric parameter and **StoreNumericDataUsingVarName** to set its final value. For clarity however, in general it is better to keep inputs and outputs separate.

Using a **VarName** parameter allows you to create a parameter whose type is not determined until runtime. The user can pass either a numeric variable or a global variable for the parameter. However this technique is obscure and should be avoided if at all possible.

You can call **VarNameToDataType** to determine which type was passed. This type flexibility is unique to **varName** parameters and may be of use in rare cases.

VarName should not be used for normal input parameters – use number instead.

Data Folder Reference Parameters

Support for data folder reference parameters was added in Igor Pro 6.20. If you use this feature, you must check Igor's version as described under **Checking Igor's Version** on page 66.

Data folder references are passed to you as data folder handles of type **DataFolderHandle**. The calling code can pass a literal data folder path (**root:MyFolder**), a **\$** followed by a string containing a data folder path, or a data folder reference variable (**DFREF**).

Data folder handles always belong to Igor. You must never dispose or directly modify a data folder handle. You operate on a data folder by passing its handle to an XOPSupport routine.

The handle for a data folder reference parameter can be **NULL**. This would happen, for example if the user passed an uninitialized data folder reference (**DFREF**) or a data folder reference whose initialization failed to your operation. You must always test a data folder handle. If it is **NULL**, do not use it. If the data folder reference is required for the operation, return a **NULL_DATAFOLDER_OP** error code.

The user can use ***** in place of a data folder reference when a data folder parameter is expected. This will result in a null data folder handle in the runtime parameter structure. If the data folder is required for the operation, return a **NULL_DATAFOLDER_OP** error code. Otherwise, interpret this to mean that the user wants default behavior.

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 exist when **Duplicate** is invoked.

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 **GetOperationDestWave** to create a destination wave as shown in the example below.

Destination Waves and Wave References

The `DataFolderAndName` type parameter is often used to allow the user to specify the name of the operation's destination wave through a `/DEST=<destwave>` flag. Traditionally, when an operation that allows you to specify a destination wave is compiled into a user-defined 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 `$"<string>"`, 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 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-defined 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", "text", "wave" and "dfref". "wave" and "dfref" require Igor Pro 6.22 or later.

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` as shown in the example below.

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` does nothing if no automatic wave reference exists.

DataFolderAndName Destination Wave Example

This section presents an example of an operation that has a destination wave parameter. It uses the `GetOperationDestWave` callback to do most of the work. `GetOperationDestWave` handles all of the following:

- Recognizing a wave reference passed as the destination wave (with Igor Pro 6.20 or later)
- Creating a new wave if the destination wave does not already exist
- Overwriting or changing the dimensions and data type of the destination wave if it already exists

In addition, the `SetOperationWaveRef` call in the sample code below sets the automatic output wave reference, if appropriate, when called from a user-defined function.

Chapter 5 — Adding Operations

The example operation is defined by this Operation Handler template:

```
SampleOp /DEST=DataFolderAndName:{dest,real}
```

This results in an operation runtime parameter structure containing:

```
int DESTFlagEncountered;  
DataFolderAndName dest;  
int DESTFlagParamsSet[1];
```

In the following code, p is a pointer to the operation runtime parameter structure.

```
waveHndl destWaveH;           // Destination wave handle.  
int destWaveRefIdentifier;     // Identifies a wave reference  
char destWaveName[MAX_OBJ_NAME+1];  
DataFolderHandle dfH;  
int dataType;  
CountInt dimensionSizes[MAX_DIMENSIONS+1];  
int options;  
int err;  
  
destWaveH = NULL;  
destWaveRefIdentifier = 0;  
  
strcpy(destWaveName, "W_SampleOp"); // Default dest wave name  
dfH = NULL;                        // Default is current data folder  
  
dataType = NT_FP64;  
MemClear(dimensionSizes, sizeof(dimensionSizes));  
dimensionSizes[ROWS] = 100;  
  
if (p->DESTFlagEncountered) {  
    strcpy(destWaveName, p->dest.name);  
    dfH = p->dest.dfH;  
    // If a wave reference was used, p->destParamsSet[0] contains  
    // information that GetOperationDestWave uses to find it.  
    destWaveRefIdentifier = p->DESTFlagParamsSet[0];  
}  
  
options = kOpDestWaveOverwriteOK | kOpDestWaveOverwriteExistingWave;  
  
err = GetOperationDestWave(dfH, destWaveName, destWaveRefIdentifier,  
    options, dimensionSizes, dataType, &destWaveH, NULL);  
if (err != 0)  
    return err;  
  
<Store output data in dest wave using destWaveH>  
WaveHandleModified(destWaveH);  
  
// Set wave reference to refer to destination wave.  
if (destWaveRefIdentifier != 0)  
    SetOperationWaveRef(tp, destWaveH, destWaveRefIdentifier);
```

The GetOperationDestWave callback creates the destination wave if it does not exist and overwrites it if it does.

The GetOperationDestWave callback was added in Igor Pro 6.20 and XOP Toolkit 5.09. When running with an older version of Igor, the XOP Toolkit emulates the behavior of GetOperationDestWave so you can call it with any version of Igor. However, the emulation can not honor wave references passed as the destination wave parameter. Consequently when running with an old version of Igor, this user-defined function code:

```
Wave/Z w = root:MyDataFolder:jack
SampleOp /DEST=w
```

will write to a wave named w in the current data folder, not root:MyDataFolder:jack.

For further details, see **GetOperationDestWave** on page 253.

DataFolderAndName Data Folder Example

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

An external operation can take a structure as a parameter. This is a technique for advanced programmers.

Structure parameters are passed as pointers to structures. These pointers always belong to Igor. You must never dispose or resize a memory block using a structure pointer but you may read and write its fields.

Igor structures are always two-byte-aligned. As the example below shows, you must use the pack pragma to make sure that your C structure is also two-byte-aligned. See **Structure Alignment** on page 209 for details on structure alignment.

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 distinctive and unlikely to be used for any other purpose.

Chapter 5 — Adding Operations

Although Igor requires that the structure passed to your operation is of the specified type, it does not guarantee that the programmer calling your operation has defined the structure properly. If the structure definition used by the Igor procedure programmer is inconsistent with the structure defined in your C code, a crash is likely to occur. You can increase the likelihood that the structure will be correct by providing an Igor procedure file that defines it. It is also a good idea to include a version field in your structure as illustrated in the next section.

In very rare cases, you might want to define an operation that takes any type of structure. For example, in implementing a sophisticated curve fitting algorithm, you might define a structure to be passed to your operation that starts with certain fields and contains other user-defined fields afterwards. Your operation accesses the known fields and passes the entire structure to a user-defined function which accesses the whole structure. If you specify void as the structure type (e.g., "void" instead of "DemoStruct"), Igor will allow the user to pass any type of structure to your operation.

In some cases you might need to know the total size of a structure passed to your operation, for example, if your operation writes a user-defined structure to a file. For this, use the extended form of the structure parameter, described under **Extended Structure Parameters** on page 90.

Wave Handles In Structures

Usually a waveHndl field in a structure is used to pass a wave reference from Igor to your XOP. However, it can also be used to pass a wave reference from your XOP to Igor.

In Igor Pro 6.20 or later, if you store a wave handle in a waveHndl field of a structure you need to inform Igor by calling **HoldWave** on the structure field. By doing this, you maintain the integrity of Igor's internal wave reference count. For details see **Wave Reference Counting** on page 128.

Data Folder Handles In Structures

Usually a DataFolderHandle field in a structure is used to pass a data folder reference from Igor to your XOP. However, it can also be used to pass a data folder reference from your XOP to Igor.

In Igor Pro 6.20 or later, if you store a data folder handle in a DataFolderHandle field of a structure you need to inform Igor by calling **HoldDataFolder** on the structure field. By doing this, you maintain the integrity of Igor's internal data folder reference count. For details see **Data Folder Reference Counting** on page 140.

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=6 "DemoStructOp structure:{sp,DemoStruct}"

#pragma pack(2)                // Igor structures are 2-byte-aligned.
#define kDemoStructVersion 1000 // 1000 means 1.000.
struct DemoStruct {
    UInt32 version;           // Structure version.
    double num;
    Handle strH;
};
typedef struct DemoStruct DemoStruct;
#pragma pack()                // Reset structure alignment.

// Operation template: DemoStructOp structure:sp

// Runtime param structure for DemoStructOp operation.
#pragma pack(2)                // Igor structures are 2-byte-aligned.
struct DemoStructOpRuntimeParams {
    // Parameters for simple main group #0.
    int spEncountered;
    DemoStruct* sp;
    int spParamsSet[1];

    // These are postamble fields that Igor sets.
    int calledFromFunction;
    int calledFromMacro;
};
typedef struct DemoStructOpRuntimeParams DemoStructOpRuntimeParams;
typedef struct DemoStructOpRuntimeParams* DemoStructOpRuntimeParamsPtr;
#pragma pack()                // Reset structure alignment.

extern "C" 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 210.

Extended Structure Parameters

An extended form of the structure parameter allows you to determine the size and structure type name of the structure passed to the XOP. The main reason for adding this feature 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,void,1}
```

Here s is the mnemonic name for a structure parameter, void signifies that your operation accepts any type of structure, 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=6 "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:

```
extern "C" 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-defined function the operation creates local variables (unless the obsolete rtGlobals=0 mode is in effect). 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 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.

File-loader operations should set the standard output variables V_flag, S_fileName, S_path and S_waveNames. These output variables must be declared when calling RegisterOperation. For example, the RegisterOperation function for SimpleLoadWave contains these statements:

```
runtimeNumVarList = "V_flag;";  
runtimeStrVarList = "S_path;S_fileName;S_waveNames;";
```

You can set the file-loader output variables by calling **SetFileLoaderOutputVariables** from your ExecuteOperation function.

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 72, 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 XOPMain function

The ExecuteOperation should be declared `extern "C"` if you are writing a C++ XOP and `static` if you are writing a C XOP. For example:

```
extern "C" int          // C++ - Generated by ParseOperationTemplate /C=6  
ExecuteXOP1(XOP1RuntimeParamsPtr p)  
  
static int            // C - Generated by ParseOperationTemplate /C=2  
ExecuteXOP1(XOP1RuntimeParamsPtr p)
```

`extern "C"` is appropriate in C++ code because the function is called from C code in Igor using C calling conventions. See **C++ Code Called From C Code** on page 218 for details. Although it is not technically correct, using `static` in a C++ XOP also works with all of the C++ compilers that we have seen.

When filling out the ExecuteOperation function, remember that you need to test string, wave and data folder reference 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 call generated by ParseOperationTemplate is complete except that you may need to add runtime numeric or string output variables as described under **Runtime Output Variables** on page 92.

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.cpp 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.

Thread-Safe External Operations

Multithreading makes it possible to utilize more than one processor on a multi-processor computer. Even on a single processor machine it allows multiple processes to run independent of one another.

As of Igor Pro 6.02, Igor supports thread-safe external operations with certain restrictions discussed below. This section tells you how to convert a normal external operation to a thread-safe operation using the XOP1 XOP as an example.

NOTE: Support for thread-safe external operations was greatly improved in Igor Pro 6.20. If your XOP implements thread-safe external operations, you can greatly simplify matters by requiring Igor Pro 6.20 or later.

Writing thread-safe code is trickier than writing regular code. For example, your operation can not write to a global variable because two instances of your operation might write different values to the global and interfere with each other's operation. Many web sites explain what "thread-safe" means and what you can and can not do in thread-safe code.

Restrictions on Thread-Safe External Operations With Older Igor Versions

In addition to the inherent restrictions on thread-safe code, there is another major restriction for thread-safe external operations when running with a version of Igor prior to 6.20: they can not do any callbacks to Igor. The reason for this is that the XOP callback mechanism was not thread-safe until Igor Pro 6.20. This restriction greatly limits what you can do from a thread-safe external operation with old versions of Igor.

The following wave access functions can be called from thread-safe external operations because they do not do callbacks when running with Igor Pro 6:

Chapter 5 — Adding Operations

WaveType, WavePoints, WaveData, WaveModDate, WaveModState, WaveModCount
MDGetWaveDimensions, MDGetWaveScaling, MDAccessNumericWaveData
MDGetNumericWavePointValue, MDSetNumericWavePointValue
MDGetDPDataFromNumericWave, MDStoreDPDataInNumericWave

All other wave access functions do callbacks to Igor and thus can not be called from thread-safe code when you are running with older versions of Igor.

The SetOperationNumVar, SetOperationStrVar and SetFileLoaderOperationOutputVariables routines are callbacks and can not be used from thread-safe external operations when you are running with older versions of Igor. This prevents setting runtime output variables such as V_flag and S_path.

Most other XOPSupport routines also do callbacks to Igor and thus can not be used from thread-safe code when you are running with older versions of Igor.

Changes for Thread-Safe External Operations

A few changes are needed to make an external operation thread-safe. In this section we illustrate these changes using the XOP1 external operation as an example. For simplicity, we require Igor Pro 6.20 or later.

Here are the steps for making the XOP1 external operation thread-safe:

1. Open the "XOP1 Template.pxp" experiment. In the procedure window, add the /TS flag to the ParseOperationTemplate command. This tells ParseOperationTemplate that your external operation is thread-safe.
2. Run the CreateXOP1StarterCode function to put new starter code in the clipboard. Go into your development system and paste the starter code into an empty text window.
3. In the XOP1.cpp source file, replace the runtime parameter structure with the new version just generated. The only difference is that the new version has an extra field:

```
UserFunctionThreadInfoPtr tp;
```

You do not have to do anything with this field but it must be there for a thread-safe external operation.

4. In the newly-generated starter code, find the call to RegisterOperation at the very bottom. Copy it to the clipboard and paste it over the old call to RegisterOperation in XOP1.cpp. The only difference is that the new code passes the kOperationIsThreadSafe flag as the last parameter when calling RegisterOperation.
5. In the XOP1.cpp file, add a check in the XOPMain function to make sure that you are running with Igor Pro 6.20 or later:

```
if (igorVersion < 620) {  
    SetXOPResult(IGOR_OBSOLETE);  
    return EXIT_FAILURE;  
}
```

6. For Macintosh, in the XOP1.r file, add threadSafeOp to the operation category:

```
resource 'XOPC' (1100) {  
    {  
        "XOP1",  
        waveOp+XOPOp+compilableOp+threadSafeOp,  
    }  
};
```

7. For Windows, in the XOP1WinCustom.rc file, add threadSafeOp to the operation category:

```
1100 XOPC  
BEGIN  
    "XOP1\0",  
    utilOp | XOPOp | compilableOp | threadSafeOp,
```

```
"\0"
END
```

When you make your own external operation thread-safe, check to make sure that it does not do anything that is not thread-safe.

Assuming that you have compiled a thread-safe version of XOP1, here is an example of how you could call it from within Igor 6.20 or later:

```
ThreadSafe Function AddOneToWave(w)
    Wave w

    XOP1 w          // Add one to wave
End

Function TestXOPInThread()
    // Create sample data
    if (!WaveExists(wave0))
        Make/O/N=10 wave0=p, wave1=p, wave2=p
    endif

    Variable numThreads = 3

    Variable mt = ThreadGroupCreate(numThreads)
    Variable i

    for(i=0; i<numThreads;)
        for(i=0; i<numThreads; i+=1)
            String name
            sprintf name, "wave%d", i
            Wave w = $name
            ThreadStart mt, i, AddOneToWave(w)      // Start thread
        endfor

        // Wait for threads to finish
        do
            Variable tgs = ThreadGroupWait(mt,100)
            while (tgs != 0)
        endfor

        Variable result = ThreadGroupRelease(mt)
        return result
    End
```

More on Multithreading

For additional discussion of multithreading in XOPs see **Multithreading** on page 167.

External Operations and Macintosh Resources

If your Macintosh external operation directly or indirectly calls `GetResource`, you must preserve the current resource fork. Call `CurResFile` to save the current resource and `UseResFile` to set the current resource to your resource fork. Do your call to `GetResource`. Finally, call `UseResFile` again to restore the current resource to what it originally was.

Accessing Macintosh resources is not thread-safe.

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 Operation Handler.
If your operation is thread-safe, make sure that you have used the threadSafeOp in the category field of the XOPC resource and that you have passed the kOperationIsThreadSafe flag to **RegisterOperation** and that your runtime parameter structure has a UserFunctionThreadInfoPtr field. Also make sure that you do not do any non-thread-safe callbacks to Igor and that your code is written in a thread-safe fashion.
- You do not have a call to **SetXOPTType**(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 77. See **Creating Starter Code** on page 72 and **Updating Starter Code** on page 93.
- You must always check string, wave and data folder handles to make sure they are not null before using them.
- You must not dispose string handles passed from Igor via your runtime parameter structure. 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. See **Understand the Difference Between a String in a Handle and a C String** on page 237 for further discussion.
- You must never dispose or directly modify a wave handle. Wave handles belong to Igor.
- You must never dispose or directly modify a data folder handle. Data folder handles belong to Igor.
- If you are updating a file loader XOP that calls **SetFileLoaderOutputVariables**, you need to change it to call **SetFileLoaderOperationOutputVariables** 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 14.

Chapter 6

Adding Functions

Overview

An XOP can add any number of external functions to Igor Pro.

An external function takes zero or more numeric, string, wave reference, data folder reference and structure parameters and returns a numeric, string, wave reference or data folder reference 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. Unlike user-defined functions, external functions can not have optional parameters.

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 $a + b \cdot \log(c \cdot x)$. This function is suitable for use in curve fitting. 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.

If the XOP has an XOPF 1100 resource, Igor loads the XOP into memory at Igor's startup time. It calls the XOP's XOPMain 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	Return type, parameter type.	Wave handle. A wave return type requires Igor Pro 6.20 or later.
DATAFOLDER_TYPE	256	Return type, parameter type.	Data folder handle. A data folder reference return or parameter type requires Igor Pro 6.20 or later.
FV_REF_TYPE	4096	Parameter type only.	Indicates pass-by-reference parameter. Requires Igor Pro 5.03 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.

If you define an external function that returns a wave, your XOP will require Igor Pro 6.20 or later. See **Checking Igor's Version** on page 66.

If you define an external function that takes a data folder reference parameter or returns a data folder reference result, your XOP will require Igor Pro 6.20 or later. See **Checking Igor's Version** on page 66.

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.cpp 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 under **Parameter and Result Types** on page 102.

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 or data folder reference parameters. Pass-by-reference is described under **Pass-By-Reference Parameters** on page 112.

FV_STRUCT_TYPE, which declares a structure parameter, must always be ORed with FV_REF_TYPE. Structure parameters are passed as pointers to structures.

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_THREADSAFE	16384	Thread-safe
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 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 as illustrated by the logfit function in XFUNC2.

External Function Parameters and Results

The XFUNC1Div routine in XFUNC1.cpp illustrates how you access the function's parameters and how you return the result:

```

#pragma pack(2)          // All Igor structures are two-byte-aligned.
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;
#pragma pack()           // Reset structure alignment.

extern "C" 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 pragma statements insure that Igor and the XOP agree on the alignment of the parameter structure. See **Structure Alignment** on page 209 for details.

The external function should be declared `extern "C"` if you are writing a C++ XOP and `static` if you are writing a C XOP. For example:

```

extern "C" int          // C++
XFUNC1Div(XFUNC1DivParams* p)

static int              // C
XFUNC1Div(XFUNC1DivParams* p)

```

`extern "C"` is appropriate in C++ code because the function is called from C code in Igor using C calling conventions. Although it is not technically correct, using `static` in a C++ XOP also works with all of the C++ compilers that we have seen.

External Function Parameters

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.

Chapter 6 — Adding Functions

Here is the correspondence between parameter types and field data types:

Parameter Type	XOPF Symbol	Field Type	Notes
Number	NT_FP64	double	
Complex number	NT_FP64 NT_CMPLX	double[2]	
String	HSTRING_TYPE	Handle	Always test for null string handles.
Wave	WAVE_TYPE	waveHndl	Always test for null wave handles.
DFREF	DATAFOLDER_TYPE	DataFolderHandle	Requires Igor Pro 6.20 or later. Always test for null data folder handles.
Structure	FV_STRUCT_TYPE FV_REF_TYPE	Pointer to structure	Always test for null structure pointers.

In the example above, the parameters are passed by value. Numeric and string parameters can also be passed by reference, as described under **Pass-By-Reference Parameters** on page 112. In that case, the structure fields would be defined as double* or Handle* and the parameter specification in the XOPF resource would be ORed with FV_REF_TYPE.

Igor can pass NULL for string, wave and DFREF handle parameters as well as structure pointer parameters. This happens when there is a bug in the calling user-defined function. Therefore you must always test these types of parameters and return an error if they are NULL.

External Function Result and Error Code

In the XFUNC1Div example above, the function result, which is returned to the calling user-defined function, is set by this line:

```
p->result = p->p1 / p->p2; // XFUNC result.
```

The error code, returned to Igor as the C function result, is set by this line:

```
return 0; // XFUNC error code
```

The error code is zero if no error occurred or a built-in Igor error code (defined in IgorErrors.h) or an XOP-defined error code (defined in the XOP's .h file with corresponding error strings in the STR# 1100 resource - see **XOP Errors** on page 57 for details).

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; as explained below, the NT_FP64 part is needed only to qualify this function as a curve-fitting function. 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.

Wave Parameters

The XOPF resource of the XFUNC2 XOP declares the wave parameter like this:

```
WAVE_TYPE | NT_FP64,    // Wave
```

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 CurveFit and FuncFit operations but 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 if the caller passes an uninitialized wave reference or a wave reference whose initialization failed.

Your function must check for a null wave handle as shown in this example from the logfit function in XFUNC2:

```

// Check that wave handle is valid
if (p->waveHandle == NULL) {
    SetNaN64 (&p->result);      // Return NaN if wave is not valid
    return NULL_WAVE_OP;        // Return error to Igor
}

```

NULL_WAVE_OP is an Igor error coded defined in IgorErrors.h.

Wave Results

The ability to define an external function that returns a wave reference was added in Igor Pro 6.20. If you use this feature, you must make sure that you are running with a recent-enough Igor version. See **Checking Igor's Version** on page 66.

Chapter 6 — Adding Functions

To define an external function that returns a wave reference, you specify `WAVE_TYPE` as the result type in the XOPF resource.

Here is code that shows how to return a wave reference. Given the name of a wave in the current data folder, it returns a wave reference or `NULL` if no such wave exists.

```
#pragma pack(2)          // All Igor structures are two-byte-aligned.
struct NameToWaveParams {
    Handle nameH;         // Name of wave in the current data folder.
    waveHndl result;
};
typedef struct NameToWaveParams NameToWaveParams;
#pragma pack()           // Reset structure alignment.

extern "C" int
NameToWave(NameToWaveParams* p)
{
    Handle nameH;
    waveHndl waveH;
    char name[MAX_OBJ_NAME+1];
    int result;

    result = 0;
    p->result = NULL;

    nameH = p->nameH;
    if (nameH == NULL)
        return USING_NULL_STRVAR;
    if (result = GetCStringFromHandle(nameH, name, sizeof(name)-1))
        return result;

    waveH = FetchWave(name);    // NULL if the wave does not exist
    p->result = waveH;

    return result;              // Error code returned to Igor
}
```

Data Folder Reference Parameters

The ability to define an external function that takes a data folder reference as a parameter was added in Igor Pro 6.20. If you use this feature, you must make sure that you are running with a recent-enough Igor version. See **Checking Igor's Version** on page 66.

A data folder reference parameter is represented by `DATAFOLDER_TYPE` in the XOPF resource.

NOTE: Igor can pass a `NULL` data folder handle to your function. This happens during the execution of a user-defined function if the caller passes an uninitialized data folder reference or a data folder reference whose initialization failed.

Your function must check for a null data folder handle as illustrated under **Data Folder Reference Example** on page 105.

Data Folder Reference Results

The ability to define an external function that returns a data folder reference was added in Igor Pro 6.20. If you use this feature, you must make sure that you are running with a recent-enough Igor version. See **Checking Igor's Version** on page 66.

To define an external function that returns a data folder reference, you specify `DATAFOLDER_TYPE` as the result type in the XOPF resource.

Data Folder Reference Example

Here is code that shows how to take a data folder reference parameter and return a data folder reference as the function result. Given a data folder reference and an index, the GetChildDF function returns a data folder reference for a child data folder or NULL if the index is out-of-bounds.

```
// Macintosh XOPF resource, in .r file.
resource 'XOPF' (1100) {
    {
        "GetChildDF",                // Function name
        F_UTIL | F_EXTERNAL,         // Function category
        DATAFOLDER_TYPE,           // Return value type = Data folder reference
        {
            DATAFOLDER_TYPE,        // Parameter 0: Data folder reference
            NT_FP64,                  // Parameter 1: Numeric index
        },
    }
};

// Windows XOPF resource, in WinCustom.rc file.
1100 XOPF
BEGIN
    "GetChildDF\0",                // Function name
    F_UTIL | F_EXTERNAL,           // Function category
    DATAFOLDER_TYPE,              // Return value type = Data folder reference
    DATAFOLDER_TYPE,              // Parameter 0: Data folder reference
    NT_FP64,                       // Parameter 1: Numeric index
    0,                             // 0 terminates list of parameter types.
    0,                             // 0 terminates the resource.
END

// C++ code in .cpp file.

#pragma pack(2)                  // All Igor structures are two-byte-aligned.
struct GetChildDFParams {
    double index;                 // Index to child data folder
    DataFolderHandle dfr;         // Parent data folder
    DataFolderHandle result;
};
typedef struct GetChildDFParams GetChildDFParams;
#pragma pack()                  // Reset structure alignment.

extern "C" int
GetChildDF(GetChildDFParams* p)
{
    DataFolderHandle parentDFH;
    DataFolderHandle childDFH;
    int index;
    int result;

    result = 0;
    p->result = NULL;

    parentDFH = p->dfr;
    if (parentDFH == NULL)
        return NULL_DATAFOLDER_OP;
    index = p->index;

    result = GetIndexedChildDataFolder(parentDFH, index, &childDFH);
    p->result = childDFH;
}
```

```
    return result;          // Error code returned to Igor
}
```

NULL_DATAFOLDER_OP is an Igor error coded defined in IgorErrors.h.

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,    // Double-precision complex 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,         // Double-precision complex 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.cpp 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 under **FUNCTION Message Versus Direct Methods** on page 114.

The return value type of `HSTRING_TYPE` says that the function returns a string. The `HSTRING_TYPE`s indicate both parameters are strings.

This function is invoked as follows:

```
String aString
aString = xstrcat0("Hello", " out there")
```

Both the `xstrcat0` and `xstrcat1` external functions wind up calling the `xstrcat` function in `XFUNC3.cpp`. `xstrcat` is defined as follows:

```

#pragma pack(2)                // All Igor structures are two-byte-aligned.
struct xstrcatParams {
    Handle str3;
    Handle str2;
    Handle result;
};
typedef struct xstrcatParams xstrcatParams;
#pragma pack()                 // Reset structure alignment.

extern "C" int
xstrcat(xstrcatParams* p)
{
    Handle str1;                // output handle
    int 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 `IgorErrors.h`. You could also use the Igor `USING_NULL_STRVAR` error code in place of `NO_INPUT_STRING`.

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 it (using **GetCStringFromHandle**) or add a null terminator to the handle. If you pass the handle back to Igor, you must remove the null terminator. See **Understand the Difference Between a String in a Handle and a C String** on page 237 for further discussion of string handles.

An external string function returns a handle as its result. It can return `NULL` in the event of an error.

NOTE: String parameter handles passed by Igor to an external function belong to the XOP. The XOP must dispose or reuse the handle.

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

Structure parameters are passed as pointers to structures. These pointers always belong to Igor. You must never dispose or resize a memory block using 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:

```
#pragma pack(2)          // All Igor structures are two-byte-aligned.

#define kF1StructureVersion 1000    // 1000 means 1.000.

struct F1Struct {          // Structure of parameter.
    UInt32 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;

#pragma pack()           // Reset structure alignment.

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 = NULL_WAVE_OP;                // 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.

The F1Struct structure contains a string handle field named strH. Unlike the case of a parameter string handle, the external operation does not receive ownership of a string handle in a structure parameter and must not dispose it.

For further important details see **Using Igor Structures as Parameters** on page 210.

Wave Handles In Structures

Usually a `waveHndl` field in a structure is used to pass a wave reference from Igor to your XOP. However, it can also be used to pass a wave reference from your XOP to Igor.

In Igor Pro 6.20 or later, if you store a wave handle in a `waveHndl` field of a structure you need to inform Igor by calling **HoldWave** on the structure field. By doing this, you maintain the integrity of Igor's internal wave reference count. For details see **Wave Reference Counting** on page 128.

Data Folder Handles In Structures

Usually a `DataFolderHandle` field in a structure is used to pass a data folder reference from Igor to your XOP. However, it can also be used to pass a data folder reference from your XOP to Igor.

In Igor Pro 6.20 or later, if you store a data folder handle in a `DataFolderHandle` field of a structure you need to inform Igor by calling **HoldDataFolder** on the structure field. By doing this, you maintain the integrity of Igor's internal data folder reference count. For details see **Data Folder Reference Counting** on page 140.

Pass-By-Reference Parameters

External functions can have pass-by-reference parameters. Only numeric, string and structure parameters can be pass-by-reference. Wave and data folder reference parameters can not be 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 provides a way for an external 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
        {
            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:

```
#pragma pack(2)           // All Igor structures are two-byte-aligned.
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;
#pragma pack()          // Reset structure alignment.
```

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];
    char buffer[512];
    int err;

    err = 0;

    // 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 this 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

The rationale for the FUNCTION message method no longer exists so the direct method is recommended for all modern external functions.

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.cpp 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 external 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 XOPMain 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 .

Direct External Functions and Macintosh Resources

If your direct Macintosh external function directly or indirectly calls GetResource, you must preserve the current resource fork. Call CurResFile to save the current resource and UseResFile to set the current resource to your resource fork. Do your call to GetResource. Finally, call UseResFile again to restore the current resource to what it originally was.

Thread-Safe External Functions

As of Igor Pro 6, Igor supports thread-safe user-defined functions. This makes it possible to utilize more than one processor on a multi-processor computer.

Igor Pro 6 also supports thread-safe external functions with certain restrictions discussed below. This section tells you how to convert a normal external function to a thread-safe function using the SimpleFit XOP as an example.

NOTE: Support for thread-safe external functions was greatly improved in Igor Pro 6.20. If your XOP implements thread-safe external functions, you can greatly simplify matters by requiring Igor Pro 6.20 or later.

Writing thread-safe code is trickier than writing regular code. For example, your function can not write to a global variable because two instances of your function might write different values to the global and interfere with each other's operation. Many web sites explain what "thread-safe" means and what you can and can not do in thread-safe code.

In addition to the inherent restrictions on thread-safe code, there is another major restriction for thread-safe external functions when running with a version of Igor prior to 6.20: they can not do any callbacks to Igor. The reason for this is that the XOP callback mechanism was not thread-safe until Igor Pro 6.20. This restriction greatly limits what you can do from a thread-safe external function with old versions of Igor.

The following wave access functions can be called from thread-safe external functions because they do not do callbacks when running with Igor Pro 6:

```
WaveType, WavePoints, WaveData, WaveModDate, WaveModState, WaveModCount
MDGetWaveDimensions, MDGetWaveScaling, MDAccessNumericWaveData
MDGetNumericWavePointValue, MDSetNumericWavePointValue
MDGetDPDataFromNumericWave, MDStoreDPDataInNumericWave
```

All other wave access functions do callbacks to Igor and thus can not be called from thread-safe code when you are running with older versions of Igor.

Most other XOPSupport routines also do callbacks to Igor and thus can not be used from thread-safe code when you are running with older versions of Igor.

Changes for Thread-Safe External Functions

A few changes are needed to make an external function thread-safe. In this section we illustrate these changes using the logfit external function from XFUNC2 as an example. For simplicity, we require Igor Pro 6.20 or later.

Here are the steps for making an external function thread-safe:

1. Make sure that your function uses the direct-call method as described under **FUNCTION Message Versus Direct Methods** on page 114.
2. Add the F_THREADSafe bit to the category field of your XOPF resource.

In the XFUNC2.r and XFUNC2.rc, change the category from:

```
F_EXP | F_EXTERNAL,
```

to

```
F_EXP | F_THREADSafe | F_EXTERNAL,
```

3. Add a check in the XOPMain function to make sure that you are running with Igor Pro 6.20 or later:

```
if (igorVersion < 620) {
    SetXOPResult(IGOR_OBSOLETE);
    return EXIT_FAILURE
}
```

4. Add a "tp" field to your external function's parameter structure. The tp field is a pointer to private Igor thread-related data that is not currently used but may be used in the future. It may be NULL. The tp field must be added just before the result field, like this:

```
typedef struct FitParams {
    double x;
    waveHndl waveHandle;
    UserFunctionThreadInfoPtr tp; // Pointer to Igor private data.
    double result;
} FitParams, *FitParamsPtr;
```

5. Check to make sure that your thread-safe external function does not do anything that is not thread-safe.

More on Multithreading

For additional discussion of multithreading in XOPs see **Multithreading** on page 167.

External Functions and Macintosh Resources

If your Macintosh external operation directly or indirectly calls `GetResource`, you must preserve the current resource fork. Call `CurResFile` to save the current resource and `UseResFile` to set the current resource to your resource fork. Do your call to `GetResource`. Finally, call `UseResFile` again to restore the current resource to what it originally was.

Accessing Macintosh resources is not thread-safe.

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 C function result which Igor looks at to see if the function encountered a fatal error.

`logfit` is defined as follows:

```
#pragma pack(2)          // All Igor structures are two-byte-aligned.
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;
#pragma pack()           // Reset structure alignment.

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 NULL_WAVE_OP;
    }

    // Check coefficient wave's numeric type.
    switch (WaveType(p->waveHandle)) {
        case NT_FP32:
            fPtr = (float*)WaveData(p->waveHandle);
            a = fPtr[0];
            b = fPtr[1];
            c = fPtr[2];
            break;
        case NT_FP64:
            dPtr = (double*)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 NULL_WAVE_OP error code is defined in IgorErrors.h. The REQUIRES_SP_OR_DP_WAVE error code is defined in XFUNC2.h with a corresponding string 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, a wave reference wave or a data folder reference wave. Logfit returns NaN if the wave passed to it is other than single- or double-precision real.

Returning a non-zero value as the C function result from an external function causes Igor to abort procedure execution and display an error alert. Therefore, you should do this only 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 NULL_WAVE_OP or REQUIRES_SP_OR_DP_WAVE.

Accessing Igor Data

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 **Macintosh Memory Management** on page 197 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 15, to get and set wave data and properties.

Routines for Accessing Waves

The XOPSupport routines for dealing with waves are described in detail under **Routines for Accessing Waves** on page 257. 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 operations and external functions that receive wave handles from Igor as parameters.

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.

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.

Wave Reference Counting

Routine	Description
HoldWave	Tells Igor that you are holding a wave handle until further notice.
ReleaseWave	Tells Igor that you are no longer holding a wave handle.

Obsolete Routines

On Mac OS 9 and before, the heap block referenced by a handle could be relocated in memory by the operating system. Because of this it was necessary at times to lock the heap block in place. This was called "locking the handle". Locking handles is unnecessary on Mac OS X and Windows.

In old XOP source code you may see the routines HLock, HUnlock, HGetState, HSetState, MoveHHi, MoveLockHandle and SetWavesStates. These routines are obsolete and can be removed.

See **Obsolete Handle Locking Routines** on page 200 for further discussion.

Example

Here is a simple example that illustrates creating a wave, filling it with values and setting a wave property. The underlined items are data types, functions, constants or structures defined in the XOPSupport library and headers.

```
static int
MakeWave0(void)
{
    waveHndl waveH;
    char waveName[MAX_OBJ_NAME+1];
    DataFolderHandle dfH;
    CountInt dimensionSizes[MAX_DIMENSIONS+1];
    float* wp;
    IndexInt p;
    double offset, delta;
    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(&waveH, waveName, dfH, dimensionSizes, NT_FP32, 1))
        return err;

    wp = (float*)WaveData(waveH); // Get a pointer to the wave data
    for(p = 0; p < 100; p++)
        *wp++ = p; // Store point number in wave
}
```

```
offset = 0; delta = .001;      // Set X scaling to .001/point
if (err = MDSetWaveScaling(waveH, 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 XOP-Support 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 and use the pointer to store a value in each element of the wave.

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, wave references and data folder references. 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 (SInt32).
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 (UInt32).
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).
WAVE_TYPE	16384	4 or 8	Wave reference (waveHndl)
DATAFOLDER_TYPE	256	4 or 8	Data folder reference (DataFolderHandle)

Any of the numeric types 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. Text, wave reference and data folder reference waves can not be complex.

Wave references and data folder references are handles. A handle is a form of pointer and therefore consists of 4 bytes when running in 32 bits and 8 bytes when running in 64 bits.

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 15. 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 **MDSetNumericWavePointValue** routines to access a single wave point at a time. For example:

```
IndexInt 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(waveH, indices, value))
    return result;
value[0] += 1;                  // Add one to real part
if (result = MDSetNumericWavePointValue(waveH, 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 to or from double-precision floating point (unless the wave is already 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:

```
BCInt numBytes;
double* dPtr;
double* dp;
int result;

numBytes = WavePoints(waveH)*sizeof(double);           // Bytes needed for copy
if (WaveType(waveH) & NT_CMPLX) // Complex data?
    numBytes *= 2;
dPtr = (double*)NewPtr(numBytes);
if (dPtr==NULL)
    return NOMEM;
if (result = MDGetDPDataFromNumericWave(waveH,dPtr)) { // Get copy.
    DisposePtr((Ptr)dPtr);
    return result;
}
dp = dPtr;
```

```
<Use dp to access wave data.>
result = MDStoreDPDataInNumericWave(waveH, 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 **Organization of Numeric Data in Memory** on page 125.

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:

```
BCInt dataOffset;
double* dp;
int result;

if (result=MDAccessNumericWaveData(waveH, kMDWaveAccessMode0, &dataOffset))
    return result;
dp = (double*)((char*)(*waveH) + 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 under **Organization of Numeric Data in Memory** on page 125. See Chapter 15 for an example illustrating the use of MDAccessNumericData.

There are two difficulties in using the direct access method.

The first 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 Wave-Access sample XOP. If you are using C++, you can use a template to handle all of the data types.

The second 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 “kMDWaveAccessMode0”. 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.

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. 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 200x200x200 double-precision wave using the release build of WaveAccess. All tests were run on a 2010-vintage MacBook Pro with an Intel i7 laptop processor running at 2.66GHz

Operating System	Direct Access	Temp Storage	Point Access
Mac OS X 10.6.3	0.018 s	0.086 s	0.214 s
Windows 7/64	0.031	0.103	0.176
Windows 7/64 Under VMWave Fusion	0.037	0.124	0.181

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:

Item	Meaning
dimensionSizes[ROWS]	Number of rows in a column.
dimensionSizes[COLUMNS]	Number of columns in a layer. Zero for 1D waves.
dimensionSizes[LAYERS]	Number of layers in a chunk. Zero for 1D and 2D waves.
dimensionSizes[CHUNKS]	Number of chunks in the wave. Zero for 1D, 2D and 3D 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. This ordering is called “column-major order”.

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 15. 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. However, if you store a binary 0 in a text wave element, Igor operations and functions, such as Print and sprintf, that treat the element as a plain-text string will treat the 0 byte as the end of the string.

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.cpp:

```
indices[1] = column;
for(row=0; row<numRows; row++) {
    indices[0] = row;
    if (result = MDGetTextWavePointValue(waveH, indices, textH))
        goto done;
    <Modify the text in textH>;
    if (result = MDSetTextWavePointValue(waveH, 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. For further discussion see **Understand the Difference Between a String in a Handle and a C String** on page 237.

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.

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. You can use the faster but more complicated **GetTextWaveData** and **SetTextWaveData** routines which are described in Chapter 15.

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:

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

Row Index X Index Data
 X Units Data Units

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
		0.0 m	2.5 m	5.0 m
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 Data Data Data
 X Units Data Units Data Units Data Units

Column Index
 Y Index and 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 your wave has.

Killing Waves

You can kill a wave using the **KillWave** XOPSupport routine. Once you have killed a wave, you must not use any wave handle referencing it.

Igor disallows the killing of a wave that is displayed in a graph or table or used in a dependency formula. If you try to kill such a wave, you will receive an error.

Igor itself, the Igor user, a user-defined procedure or another XOP can also kill a wave. This can happen only if you return to Igor or if you do a callback to Igor. Most callbacks will not kill any waves but some, such as **XOPCommand** or **CallFunction**, can call user procedures which can do anything.

If a wave is killed and you subsequently use a wave handle referencing the killed wave, a crash is likely to occur.

You can prevent the killing of a wave by responding to the OBJINUSE message.

Preventing the Killing of a Wave

As Igor disallows the killing of a wave displayed in a graph, which would undermine the graph, you may want to disallow the killing of a wave that you are using over an indefinite period, such as a wave into which you are acquiring data. You can do this by responding to the OBJINUSE message.

Before killing a wave, Igor sends the OBJINUSE message to each XOP. If the XOP reports that the wave is in use, Igor refuses to kill it.

The OBJINUSE message is sent to XOPs only from the main Igor thread. It is not sent when an Igor preemptive thread is executing because the XOP message protocol is not threadsafe.

Responding to the OBJINUSE message prevents a wave from being *killed* while another technique, wave reference counting discussed on page 128, prevents it from being *deleted*. Wave reference counting is especially important if you are doing multithreading. It is discussed in the next section.

If your code may deal with waves from both the main Igor thread and a preemptive Igor thread, and if there is any chance that the wave may be killed while you are holding a reference to it, you should use both the OBJINUSE message and wave reference counting.

When a new experiment is loaded or Igor quits, all objects are killed, whether they are in use or not. Therefore, if your XOP holds a handle to a wave, it must clear that handle and cease accessing the wave when Igor sends you the NEW or CLEANUP messages.

See page 55 for further discussion of the OBJINUSE message.

Wave Reference Counting

The information in this section is needed only for advanced XOPs.

Internally in Igor, a wave reference is a memory location that contains a waveHndl.

An Igor user can create a wave reference by executing a WAVE statement in a user-defined function, by passing a wave as a parameter from one user-defined function to another, by setting a WAVE field in a structure and by setting an element of a wave wave. (A wave wave is a wave containing wave references created by Make/WAVE.)

An XOP holds a reference to a wave in a waveHndl variable.

If the underlying wave is deleted and Igor or an XOP subsequently attempts to use the waveHndl variable, a crash results. A technique is needed to prevent such crashes.

Internally Igor uses reference counting to determine when there are no more references to a wave in which case it is safe to delete it. The **HoldWave** and **ReleaseWave** XOPSupport routines allow XOPs to participate in reference counting.

When a regular (non-free) wave is created, Igor links the wave into its data folder. This link constitutes a reference to the wave, so Igor sets the wave's reference count to 1.

By contrast, a free wave is not part of any data folder. Therefore its reference count at creation is 0.

HoldWave increments the reference count of a wave. ReleaseWave decrements it and, if the count goes to zero or below, deletes the wave.

HoldWave and **ReleaseWave** were added in Igor Pro 6.20. If you call them with an earlier version of Igor, they will return IGOR_OBSOLETE and do nothing.

Killing Versus Deleting a Wave

There is a subtle distinction between *killing* a wave and *deleting* it. "Killing" means unlinking a wave from its data folder so that it no longer appears in the data hierarchy. "Deleting" means freeing the memory holding the wave's structure and data. Though we usually think of deleting as an integral part of killing, a wave can be killed without being immediately deleted.

Understanding this distinction helps to understand the purpose and mechanics of wave reference counting. This example user-defined function may clarify the distinction:

```
Function Test()
    String name = "jack"

    // Igor creates wave jack and sets the reference count to 0.
    // Igor then links jack into the current data folder.
    // The link constitutes a reference to the wave so Igor
    // increments the reference count to 1.
    Make/O/N=10 $name = p

    // We create another reference to the wave and the reference count
    // is incremented to 2.
    Wave w = $name

    Print WaveExists(w), WaveType(w)    // Verify that the wave is alive

    // We create another reference to the wave and the reference count
    // is incremented to 3.
    Wave w2 = $name

    // This statement unlinks the wave from the data folder AND clears w
    // but w2 still references the wave. The unlinking and the clearing
    // both decrement the reference count so it winds up at 1.
    // Because it is not zero, the wave is not deleted.
    KillWaves w

    // This statement prints "0,0" because a killed wave is considered
    // to not exist even though it has not yet been deleted.
    Print WaveExists(w2), WaveType(w2)
End    // w2 goes out of scope. Igor decrements the reference count to 0
      // and the wave is deleted.
```

Reference counting prevents the untimely deletion of a wave but does not prevent it from being killed. The OBJINUSE message, discussed under **Preventing the Killing of a Wave** on page 128, prevents a wave from being killed.

If your code may deal with waves from both the main Igor thread and a preemptive Igor thread, and if there is any chance that the wave may be killed or deleted while you are holding a reference to it, you should use both the OBJINUSE message and wave reference counting.

XOPs and Wave Reference Counting

An XOP may "hold" a wave handle. "Hold" means that you are storing a waveHndl over a period of time during which Igor could possibly delete it. **HoldWave** allows you to tell Igor that the wave should not be deleted until you subsequently call **ReleaseWave**.

For example, a data acquisition XOP that stores data in a wave during IDLE messages would typically store the wave handle in a global variable and write to that wave each time it receives an IDLE message. It is important that Igor not delete the wave while the XOP is holding a reference to it.

Chapter 7 — Accessing Igor Data

This example illustrates the use of HoldWave/ReleaseWave in a simple data acquisition application:

```
static waveHndl gWaveH = NULL;    // Global to hold data acq wave

int
StartAcq()                        // Called from your StartAcq operation or function
{
    gWaveH = FetchWave("jack");
    if (gWaveH == NULL)
        return NOWAV;

    // Tell Igor we are holding this wave handle
    HoldWave(gWaveH);

    . . .
}

int
DoAcq()                          // Called from IDLE message
{
    if (gWaveH == NULL)
        return NOWAV; // gWaveH should not be NULL during data acq
    . . .             // Acquire data into the wave
}

int
EndAcq()                        // Called from your EndAcq operation or function
{
    if (gWaveH == NULL)
        return DATA_ACQ_NOT_RUNNING;

    // Tell Igor we are no longer holding a reference to the wave
    ReleaseWave(&gWaveH); // NOTE: Sets gWaveH to NULL
    . . .
}
```

Note that ReleaseWave sets your wave handle (gWaveH in this example) to NULL and it is no longer valid.

You also should call HoldWave and ReleaseWave if you are doing a callback to Igor and the callback could possibly delete the wave - for example, if you are doing an **XOPCommand** callback or a **CallFunction** callback to run a user-defined function that could kill the wave. This is needed because you are holding a reference to the wave over a period of time in which it could be deleted.

If you are just using the wave handle temporarily during the execution of your external function or operation and you make no calls that could potentially delete the wave then you do not need to and should not call HoldWave and ReleaseWave.

If you are not sure if the wave could be deleted while you are holding it, call HoldWave/ReleaseWave. The extra pair of calls will do no harm besides the time they take.

Structure Parameters and Wave Reference Counting

Normally you will call both HoldWave and ReleaseWave yourself but there is an infrequently encountered case where Igor will call ReleaseWave instead of you.

If you define an external function or operation with a structure parameter and the structure contains a WAVE field and your XOP returns a wave handle to the calling user-defined function by setting the WAVE field, you must call HoldWave on the wave because the WAVE field holds a new reference to the wave.

You then return to the calling user-defined function. When it finishes and the structure goes out of scope, Igor calls ReleaseWave on the structure WAVE field. In this case, you are passing responsibility for refer-

ence counting to the calling user-defined function which is then responsible for matching your HoldWave with its own ReleaseWave.

If the WAVE field previously held a wave reference then Igor will have previously called HoldWave on the wave. By storing into the field, you are eliminating a reference to whatever wave was previously referenced so you must release the previous wave reference first.

Putting it all together, your code would look like this:

```
// Structure of parameter to external operation or function.
struct WaveRefExampleStruct {
    waveHndl waveH;          // Input wave and output wave reference
};
typedef struct WaveRefExampleStruct WaveRefExampleStruct;

// Parameter structure sent from Igor to XOP
struct WaveRefExampleParam {
    WaveRefExampleStruct* sp; // Pointer to a user-defined structure
    double result;
};
typedef struct WaveRefExampleParam WaveRefExampleParam;

// Called from Igor by: result = WaveRefExample(<structure>)
int WaveRefExample(struct WaveRefExampleParam* p)
{
    struct WaveRefExampleStruct* sp;
    waveHndl newWaveH;
    int err = 0;

    sp = p->sp;
    if (sp == NULL) {
        err = EXPECT_STRUCT;
        goto done;
    }

    newWaveH = FetchWave("jack"); // NULL if jack does not exist

    if (sp->waveH != newWaveH) { // No release or hold needed if same
        // We are going to store a wave handle in sp->waveH. This constitutes
        // a new reference to that wave. But there may already be a wave handle
        // in sp->waveH. In that case, storing into sp->waveH eliminates
        // a reference to the original wave so we must release it.

        // Sets sp->waveH to NULL. Does nothing if sp->waveH is already NULL.
        ReleaseWave(&sp->waveH); // Release original wave if any

        sp->waveH = newWaveH; // NULL if jack does not exist

        // Structure field constitutes a new reference to wave jack
        if (sp->waveH != NULL)
            HoldWave(sp->waveH); // Hold wave referenced by structure field
    }

done:
    p->result = err;

    // We now return to the calling user-defined function which is
    // responsible for calling ReleaseWave on sp->waveH.

    return err;
}
```

New Experiment and Wave Reference Counting

When an experiment is closed, either through New Experiment, opening another experiment or quitting Igor, all waves are killed whether you are holding them or not. Consequently, you must do something like this when your XOPEntry routine receives the NEW or CLEANUP message:

```
case NEW:           // New experiment
case CLEANUP:       // About to quit
    // Wave about to be killed. Make sure global reference is NULL.
    if (gWaveH != NULL)
        ReleaseWave(&gWaveH); // Sets gWaveH to NULL
```

Wave Reference Waves

A wave reference wave is a wave that contains references to other waves in the form of waveHndls. Such a wave can be used, for example, to pass a variable length list of waves to a user-defined function. Wave reference waves were added to Igor in Igor Pro 6.10. The contents of a given element of a wave reference wave is a wave handle or NULL.

You can create a wave reference wave by passing WAVE_TYPE as the dataType parameter to **MakeWave** and **MDMakeWave**.

Igor does not permit making complex wave reference waves.

Use the **HoldWave** callback routine after storing a wave handle in a wave reference wave element. To clear an element, use **ReleaseWave**. Using HoldWave and ReleaseWave is necessary to maintain the integrity of Igor's internal reference counting mechanism.

Data Folder Reference Waves

A data folder reference wave is a wave that contains references to data folders in the form of DataFolderHandles. Such a wave can be used, for example, to pass a variable length list of data folders to a user-defined function. Data folder reference waves were added to Igor in Igor Pro 6.10. The contents of a given element of a data folder reference wave is a data folder handle or NULL.

You can create a data folder reference wave by passing DATAFOLDER_TYPE as the dataType parameter to **MakeWave** and **MDMakeWave**.

Igor does not permit making complex data folder reference waves.

Use the **HoldDataFolder** callback routine after storing a data folder handle in a data folder reference wave element. To clear an element, use **ReleaseDataFolder**. Using HoldDataFolder and ReleaseDataFolder is necessary to maintain the integrity of Igor's internal reference counting mechanism.

Free Waves

Free waves are temporary waves that are not part of Igor's data hierarchy. They are created in user-defined functions and cease to exist when the function returns. You should create a free wave when the user requests it through a /FREE flag or some other parameter or when you need a wave for your own temporary use.

Free waves were added in Igor Pro 6.10 but to make a free wave from an XOP requires Igor Pro 6.20 or later.

You can make a free wave using **MDMakeWave** but it is more common to make a free wave as a destination wave using **GetOperationDestWave**. In either case, you must check that you are running with Igor Pro 6.20 before trying to create a free wave.

Free Waves and Wave Reference Counting

The lifetime of a free wave, like that of a regular (non-free) wave, is controlled by its reference count. See **Wave Reference Counting** on page 128 for background information.

When a regular wave is created, Igor links the wave into its data folder. This link constitutes a reference to the wave, so Igor sets the wave's reference count to 1.

By contrast, a free wave is not part of any data folder. Therefore its reference count at creation is 0.

When you create a free wave, you must call **HoldWave** on it to increment the reference count, and then, when you are finished with it, call **ReleaseWave** to decrement the reference count.

Igor deletes the wave (free or regular) when its reference count is decremented to 0 or below. The reference count is decremented when you call **ReleaseWave** or **KillWave**. It can also be decremented by Igor, the user or another XOP if they have access to the free wave's handle.

Using reference counting, through **HoldWave** and **ReleaseWave** rather than **KillWave**, is the preferred approach for free waves. However, **KillWave** does internally call **ReleaseWave** which decrements the wave's reference count and, if the count goes to zero or below, deletes it.

You can determine if a wave is a free wave by calling **GetWavesDataFolder**. If it returns a NULL data folder handle then the wave is free.

Free Wave Reference Counting Examples

To help you think about free waves and reference counting, here are some examples. "wRefCount" is used to represent Igor's internal reference count for a wave.

In the first example, a user-defined function creates a free wave and passes it to an XOP which uses it only during the external function call. The external function call does not do any callbacks to Igor that could cause the reference count to be decremented. The XOP does not need to call **HoldWave/ReleaseWave**.

```
Function Example1()
    // A free wave is created. wRefCount = 0.
    // A reference to the wave is stored in w. wRefCount = 1.
    Wave w = NewFreeWave(10,2)

    // The free wave is passed to an XOP which uses it only momentarily.
    XOPFunction1(w)
End    // w goes out of scope, wRefCount is decremented to 0, wave is deleted.
```

Although there is no need for the XOP to call **HoldWave/ReleaseWave**, doing so would not cause a problem other than taking some time.

In the next example, a user-defined function creates a free wave and passes it to an XOP which copies it to a global variable and then returns to Igor. The XOP later accesses the wave through the global variable. Because it is holding a wave reference over a period of time during which it could be deleted, the XOP must call **HoldWave**.

```
Function Example2()
    // A free wave is created. wRefCount = 0.
    // A reference to the wave is stored in w. wRefCount = 1.
    Wave w = NewFreeWave(10,2)

    // The free wave is passed to an XOP which stores it in a global waveHndl.
    // The XOP calls HoldWave on it. wRefCount = 2.
    XOPFunction2(w)
End    // w goes out of scope, wRefCount is decremented to 1.
      // Since wRefCount > 0, the wave is not deleted.

...    // Time passes during which the XOP receives IDLE messages.
      // The XOP access the wave during these messages.

// The XOP, finished with the wave, calls ReleaseWave on the global waveHndl.
// wRefCount goes to 0 and the wave is deleted.
```

Chapter 7 — Accessing Igor Data

Storing the wave handle in the global variable constitutes holding a reference to the wave. Therefore the XOP must call `HoldWave`. If it did not, when the `Example2` user-defined function returned, the wave would be deleted, leaving the XOP holding an invalid wave handle.

In the next example, the XOP creates a free wave for use during the invocation of an external function only.

```
// Create free wave with wRefCount = 0.
waveHndl freeWaveH;
MDMakeWave(&freeWaveH, "free", (DataFolderHandle)-1,
           dimensionSizes, NT_FP64, 1);
HoldWave(freeWaveH);           // wRefCount = 1.
<Use freeWaveH>
ReleaseWave(&freeWaveH);       // wRefCount = 0 and wave is deleted.
```

If the XOP does nothing that could possibly delete the wave, the call to `HoldWave` is not strictly required but it is recommended since the code could later be changed such that the wave could be deleted.

If the XOP did not call `ReleaseWave`, the free wave would never be deleted and this would constitute a leak.

In the final example, the XOP creates a free wave and, using **CallFunction**, passes it to a user-defined function.

```
// Create free wave with wRefCount = 0.
waveHndl freeWaveH;
MDMakeWave(&freeWaveH, "free", (DataFolderHandle)-1, dims, NT_FP64, 1);
HoldWave(freeWaveH);           // wRefCount = 1

<Pass freeWaveH to a user-defined function via CallFunction>
Function UserFunction(w)
    Wave w    // Creation of wave parameter. wRefCount = 2.
    ...
End          // w goes out of scope. wRefCount = 1.

<XOP uses freeWaveH>

ReleaseWave(&freeWaveH);       // wRefCount = 0 and wave is deleted.
```

In this case, if the XOP did not call `HoldWave`, `wRefCount` would go to 0 and the wave would be deleted prematurely when the user-defined function returned.

Numeric and String Variables

This section describes 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 92. Except for that 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. If you pass the handle back to Igor, you must remove the null terminator. For further discussion see **Understand the Difference Between a String in a Handle and a C String** on page 237.

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.

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 under **Routines for Accessing Variables** on page 289. Here is a summary of the commonly-used routines.

Creating Variables

Routine	Description
Variable	Creates an Igor numeric or string variable. This routine is antiquated. The following routines are recommended.
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.
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.
SetFileLoaderOperationOutputVariables	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 92.

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, string variables and other data folders 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 data folder 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 a data folder in a `DataFolderHandle`. Each XOPSupport routine that deals with data folders returns a `DataFolderHandle` to you and/or requires a `DataFolderHandle` from you. You have no way of knowing and do not need to know the internal details of `DataFolderHandles`. They are black boxes. You merely receive them from Igor and pass them back to Igor. `DataFolderHandles` belong to Igor so you must not modify or dispose them.

When you get a `DataFolderHandle` from Igor, you typically use it momentarily and do not retain it. For most applications, you should avoid storing it long-term because it will become invalid if the user kills the data 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 15. 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
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.
GetDataFolderByIDNumber	Returns a handle to a data folder identified by an ID number that you previously obtained from GetDataFolderIDNumber .

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
GetNumDataFolderObjects	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 15. Advanced programmers may want to use data folder reference counting, described in the next section.

Data Folder Reference Counting

An XOP may "hold" a data folder handle. "Hold" means that you are storing a DataFolderHandle over a period of time during which Igor could possibly delete it.

Most XOPs have no need to hold data folders. The information in this section is needed only for advanced XOPs.

Igor Pro 6 and later implement data folder reference counting. Igor keeps a reference count for each data folder. This count is used to determine if a data folder is in use.

As of Igor Pro 6.20, XOPs can participate in data folder reference counting. If your XOP obtains a data folder handle and holds it after the XOP returns to Igor, you should call **HoldDataFolder** to increment Igor's internal reference count for that data folder. When you no longer need to access the data folder, you must call

ReleaseDataFolder. ReleaseDataFolder decrements the data folder reference count and, if the count reaches zero, deletes the data folder.

For example, a data acquisition XOP that stores data in a data folder during IDLE messages might store the data folder handle in a global variable and write to that data folder each time it receives an IDLE message. It is important that Igor not delete a data folder while the XOP is holding a reference to it. To prevent the deletion, the XOP must call HoldDataFolder and then ReleaseDataFolder when you are finished with it.

You also should call HoldDataFolder and ReleaseDataFolder if you are doing a callback to Igor and the callback could possibly delete the data folder - for example, if you are doing an **XOPCommand** callback or a **CallFunction** callback to run a user-defined function that could kill the data folder. This constitutes "holding the data folder" because you are holding a reference to the data folder over a period of time in which it could be deleted. Therefore you should call HoldDataFolder to tell Igor you are using it and ReleaseDataFolder when you are no longer using it.

If you are just using the data folder handle temporarily during the execution of your external function or operation and you make no calls that could potentially delete the data folder then you do not need to and should not call HoldDataFolder and ReleaseDataFolder.

The code for holding and releasing a data folder is very similar to the code for holding and releasing a wave. See **Wave Reference Counting** on page 128 for sample code.

If you are running with an Igor version prior to 6.20, HoldDataFolder does nothing and returns IGOR_OBSOLETE. If you must run with a version earlier than 6.20, you have no way to prevent the killing of a data folder. Therefore you should use a different strategy. See **GetDataFolderIDNumber** and **GetDataFolderByIDNumber** for further information.

Structure Parameters and Data Folder Reference Counting

Once you have called HoldDataFolder on a data folder, Igor will not delete it until ReleaseDataFolder has been called on it.

Normally you will call ReleaseDataFolder yourself but there is an infrequently-encountered case where Igor will call ReleaseDataFolder on it.

If you define an external function or operation with a structure parameter and the structure contains a DFREF field and your XOP returns a data folder handle to the calling user-defined function by setting the DFREF field, you must call HoldDataFolder on the data folder because the DFREF field holds a new reference to the data folder.

You then return to the calling user-defined function. When it finishes and the structure goes out of scope, Igor calls ReleaseDataFolder on the structure DFREF field. In this case, you are passing responsibility for reference counting to the calling user-defined function which is then responsible for matching your HoldDataFolder with its own ReleaseDataFolder.

If the DFREF field previously held a data folder reference then Igor will have previously called HoldDataFolder on the data folder. By storing into the field, you are eliminating a reference to whatever data folder was previously referenced so you must release the previous data folder reference first.

See **Structure Parameters and Wave Reference Counting** on page 130 for code showing how to put this all together in the case of a wave reference structure field. The pattern is the same for a data folder reference field.

New Experiment and Data Folder Reference Counting

When an experiment is closed, either through New Experiment, opening another experiment or quitting Igor, all data folders are killed whether you are holding them or not. Consequently, you must release any data folder that you are holding by calling ReleaseDataFolder when you receive the NEW or CLEANUP messages from Igor.

Adding Menus and Menu Items

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 contain none of the resources discussed below. If you are writing an XOP of this kind, you don't need to read this chapter.

A more elaborate XOP may need to add:

- A single menu item to a built-in 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.
- A single menu item to multiple built-in Igor menus
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.
- A menu item with an attached submenu
For example, an XOP that adds a family of number-crunching operations to Igor might want to add a submenu to Igor's Analysis menu.
- A menu in Igor's main menu bar
For example, an XOP that adds significant functionality to Igor, such as a data acquisition system, might want to add a main menu bar menu.

An XOP adds menu items, submenus and main menu bar menus by defining resources that describe what is to be added and by responding to menu-related messages sent by Igor.

The MenuXOP1 sample XOP illustrates all of the scenarios listed 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** on page 146. 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 add a single menu item.

XOPSupport Menu Routines

This section gives a brief description of menu support routines provided by the XOPSupport library. You need to know this only if your XOP adds, removes, or modifies menus or menu items.

Chapter 8 — Adding Menus and Menu Items

Most of these routines were added in XOP Toolkit 6.40 to replace ancient routines that were based on the Macintosh Menu Manager. Although added in XOP Toolkit 6.40, they work with all versions of Igor with minor exceptions.

If you are updating an existing XOP to use the routines added in XOP Toolkit 6.40, see “XOP Menu Compatibility” in the XOPMenus.c file for information that will help you do the update. See also **Changes To XOP Menu Support** on page 486.

Summary of XOPSupport Menu Routines

Here are the XOPSupport menu routines:

Routine	What It Does
ResourceToActualMenuID	Given the ID of a MENU resource in the XOP's resource fork, returns the actual menu ID of that menu in memory.
ActualToResourceMenuID	Given the ID of a menu in memory, returns the resource ID of the MENU resource in the XOP's resource fork.
ResourceToActualItem	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.
ActualToResourceItem	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 specification in the XMI1 resource in the XOP's resource fork for that item.
SetIgorMenuItem	Enables or disables a built-in Igor menu item.
XOPActualMenuIDToMenuRef	Given the actual menu ID of an XOP menu in memory, returns a menu reference for the XOP menu.
XOPResourceMenuIDToMenuRef	Given the ID of a MENU resource in the XOP's resource fork, returns a menu reference for that menu.
XOPGetMenuInfo	Returns information about a menu.
XOPCountMenuItems	Returns the number of items in a menu.
XOPShowMainMenu	Makes a menu appear in the menu bar if it was previously hidden.
XOPHideMainMenu	Removes the menu from the menu bar if it was previously showing.
XOPGetMenuItemInfo	Returns information about a menu item.
XOPGetMenuItemText	Returns the text from an XOP menu item.
XOPSetMenuItemText	Sets the text of an XOP menu item.
XOPAppendMenuItem	Adds a menu item to the end of an XOP menu.
XOPInsertMenuItem	Inserts a menu item in an XOP menu.
XOPDeleteMenuItem	Removes a menu item from an XOP menu.
XOPDeleteMenuItemRange	Removes a range of menu items from from an XOP menu.
XOPEnableMenuItem	Enables a menu item.
XOPDisableMenuItem	Disables a menu item.
XOPCheckMenuItem	Checks or unchecks a menu item.
XOPFillMenu	Inserts multiple items into an XOP menu.

Routine	What It Does
XOPFillMenuNoMeta	Inserts multiple items into an XOP menu.
XOPFillWaveMenu	Inserts multiple wave names into an XOP menu.
XOPFillPathMenu	Inserts multiple symbolic path names into an XOP menu.
XOPFillWinMenu	Inserts multiple window names into an XOP menu.

XOPMenuRefs

An XOPMenuRef is a pointer managed by Igor. Most XOPSupport menu routines take an XOPMenuRef as a parameter to identify the menu of interest. You obtain an XOPMenuRef for a menu that your XOP added to Igor by calling **XOPResourceMenuIDToMenuRef**.

When running with Igor Pro 6 on Macintosh an XOPMenuRef is a Macintosh Menu Manager menu handle. When running with Igor Pro 6 on Windows an XOPMenuRef is a handle created internally by Igor that emulates a Macintosh menu handle. When running with Igor Pro 7 an XOPMenuRef a pointer to an Igor C++ object. Each version of Igor handles XOPMenuRefs in its own way and you do not need to be concerned about the details.

Here is an example that illustrates the use of XOPMenuRefs:

```
XOPMenuRef menuRef;
menuRef = XOPResourceMenuIDToMenuRef(100);           // Get XOP's menu reference.
if (menuRef != NULL) {                               // Always test before using.
    XOPDisableItem(menuRef, 1);                       // Disable the first item.
    XOPSetMenuItemText(menuRef, 2, "Capture Image"); // Set text of second item.
}
```

XOPResourceMenuIDToMenuRef is an XOPSupport routine and is described under **Getting Your Menu Reference** on page 149.

As shown you should always test a menu reference for NULL before using it. This will prevent a crash in the event of a bug in Igor or in your XOP or in the event of unforeseen changes in future versions of Igor.

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.

Here are the XOP resources that determine what an XOP adds:

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.

Chapter 8 — Adding Menus and Menu Items

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
    100,                    // menuResID: Menu resource ID is 100.
    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 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 using **XOPShowMainMenu** and **XOPHideMainMenu**. 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,                        // menuItem: 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,                            // menuItem: 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 `menuItem`, 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 in Igor Menus** on page 150.

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:

```
resource 'XSM1' (1100) {           // Macintosh, in MenuXOP1.r.
    {
        101,                      // subMenuResID: Add submenu with resource ID 101
        100,                      // mainMenuResID: to menu with resource ID 100
        1,                        // mainMenuItemNum: to item 1 of main menu
    }
};

1100 XSM1                          // Windows, in MenuXOP1WinCustom.rc.
BEGIN
    101,                          // subMenuResID: Add submenu with resource ID 101
    100,                          // mainMenuResID: to menu with resource ID 100
    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.

Chapter 8 — Adding Menus and Menu Items

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 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:

```
// From VDT.r
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.

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.

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, **ResourceToActualMenuID** and **ActualToResourceMenuID**:

```
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 reference. You can get the menu reference by calling **XOPResourceMenuIDToMenuRef** as described in the next section.

Getting Your Menu Reference

To actually do anything with your menu, such as enable or disable an item, you need to get a menu reference. The simplest way to do this is to call **XOPResourceMenuIDToMenuRef**. You pass it the resource menu ID for your menu and it returns the menu reference for the menu. This works for any menus that your XOP adds to Igor, including main menu bar menus and submenus.

It is also possible to use **XOPActualMenuIDToMenuRef** instead of **XOPResourceMenuIDToMenuRef**. You would first need to call **ResourceMenuIDToActualID** and then call **XOPActualMenuIDToMenuRef**. However there is no reason to do this. Use **XOPResourceMenuIDToMenuRef** instead.

When you obtain a menu reference you should always test it for NULL before using it. This will prevent a crash in the event of a bug in Igor or in your XOP or in the event of unforeseen changes in future versions of Igor.

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.

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 by sending the MENUENABLE message.

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. 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 menu-item-specific 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 in Igor Menus

When you receive the MENUENABLE message you can enable or disable menu items added to Igor menus by your XOP using the **XOPEnableMenuItem** or **XOPDisableMenuItem** menu routines. These routines require the menu reference for the menu, which you can obtain from **XOPActualMenuIDToMenuRef**.

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 exist 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
int 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 in response to the `MENUENABLE` message:

```
if (IsXOPWindowActive((xopWindowRef)) // Is XOP window active?
    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:

```
if (IsXOPWindowActive((xopWindowRef)) // Is XOP window active?
    SetIgorMenuItem(UNDO, 1, "Undo XOP-Specific Action", 0);
```

Call **SetIgorMenuItem** only if your XOP window is the active window.

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 to set 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)
{
    .
    .
    if (IsXOPWindowActive((xopWindowRef)) { // Is XOP window active?
        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 user 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.

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. The original Macintosh Menu Manager restricted the range of submenu IDs to less than 256. This is no longer the case but the range 200-219 is traditionally used for submenus.

Chapter 9

Adding Windows

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.

NOTE: This chapter discusses adding windows to Igor Pro 6. Everything is different in Igor Pro 7 and later. Consequently, we recommend that you not write new code for adding windows to Igor using the techniques described in this chapter.

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 sample 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 "Gizmo" 3D plotting XOP.

On Macintosh, Igor is based on Apple's Carbon API, an old API that has its roots in the pre-Mac OS X era. An advanced programmer may be able to add a window using Apple's newer Cocoa API, but this chapter is written under the assumption that you are using the Carbon API.

Adding a window on Macintosh is quite different from adding a window on Windows. The main difference is that a Macintosh XOP receives window-related messages (e.g., CLICK, ACTIVATE, UPDATE) from Igor while a Windows XOP receives analogous 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.cpp and WindowXOP1Win.cpp, thus allowing WindowXOP1.cpp 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 WindowRef 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 Carbon GetNewWindow toolbox routine. However, GetXOPWindow sets the win-

dowKind 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, GetXOPWindow is called from CreateXOPWindow.

When you are finished with your window, use the Carbon 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 **ArrowCursor**, **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 156. The files WindowXOP1.cpp and WindowXOP1Win.cpp 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 MENU-ENABLE message and your window is the top window you need to set these items appropriately. The section **Enabling and Disabling Menu Items** on page 150 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.cpp 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 XOP's XOPMain 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 SendWinMessageToIgor function.

Please refer to the WindowXOP1.cpp and WindowXOP1Win.cpp files while reading the following discussion of these routines.

CreateXOPWindowClass

CreateXOPWindowClass is called from the XOP's XOPMain function in WindowXOP1.cpp and is defined in WindowXOP1Win.cpp. 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 CLEANUP 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 CreateXOPWindowClass 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 XOPMain function in WindowXOP1.cpp and is defined in WindowXOP1Win.cpp. 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.cpp.

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, your window procedure 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 CreateXOPWindowClass. It is defined in WindowXOP1Win.cpp.

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 XOP's 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_LBUTTONDOWN, 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 VDT2 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 WindowRef on Macintosh and an HWND on 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 VDT2 XOP needs to handle keyboard messages so that it can send characters that the user types to the serial port. To achieve this, the VDT2 XOP uses subclassing - a technique described in the Windows API documentation.

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. See **How Igor Calls an XOP** on page 53 for a discussion of the basic XOP message protocol.

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.

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 `WindowRef`. On Windows, it is an `HWND`.

ACTIVATE	<p>Tells your XOP that it needs to activate or deactivate a window.</p> <p>Argument 0: <code>XOP_WINDOW_REF</code> for window to activate.</p> <p>Argument 1: Modifiers field from activate EventRecord.</p> <p>Result: None.</p> <p>Windows Note: Igor does not send the ACTIVATE message to Windows XOPs. Instead, the XOP's window procedure receives the <code>WM_MDIActivate</code> message directly from the Windows OS.</p>
UPDATE	<p>Tells your XOP that it needs to update a window.</p> <p>Argument 0: <code>XOP_WINDOW_REF</code> for window to update.</p> <p>Result: None.</p> <p>Windows Note: Igor does not send the UPDATE message to Windows XOPs. Instead, the XOP's window procedure receives the <code>WM_PAINT</code> message directly from the Windows OS.</p>
GROW	<p>Tells your XOP that it needs to change the size of its window.</p> <p>Argument 0: <code>XOP_WINDOW_REF</code> for the window.</p> <p>Argument 1: Size to set the window to.</p> <p>Result: None.</p> <p>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.</p> <p>Windows Note: Igor does not send the GROW message to Windows XOPs. Instead, the XOP's window procedure receives the <code>WM_SIZE</code> message directly from the Windows OS.</p>
SETGROW	<p>Asks your XOP for the minimum allowable size for window.</p> <p>Argument 0: <code>XOP_WINDOW_REF</code>.</p> <p>Argument 1: Pointer to a Rect to receive grow limits.</p> <p>Result: None.</p> <p>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.</p> <p>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.</p>
CLOSE	<p>Tells your XOP that the user wants to close its window.</p> <p>Argument 0: <code>XOP_WINDOW_REF</code>.</p> <p>Argument 1: Close type.</p> <p>Result: None.</p> <p>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,</p>

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.

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.

XOP_HIDE_WINDOW

Tells you to hide your XOP window.

Argument 0: XOP_WINDOW_REF identifying your window.
Result: 1 if you hide your window, 0 if you did not hide it.

This message is sent by Igor Pro 6.00 or later. For example, you will receive this if the user executes "DoWindow/HIDE=1" on your window.

You should respond by calling `XOPHideWindow`. See also **Activating, Showing and Hiding Windows** on page 164.

XOP_SHOW_WINDOW Tells you to show your XOP window.

Argument 0: `XOP_WINDOW_REF` identifying your window.

Result: 1 if you showed your window, 0 if you did not show it.

This message is sent by Igor Pro 6.00 or later. For example, you will receive this if the user executes "DoWindow/HIDE=0" on your window.

You should respond by calling `XOPShowWindow`. See also **Activating, Showing and Hiding Windows** on page 164.

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, **Adding Menus and Menu Items**.

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 **SetXOPWindowIgorPositionAndState**.

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, **Adding Menus and Menu Items**.

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 **SetXOPWindowIgorPositionAndState**.

If you want to support this message, you must enable the Retrieve Window menu item by calling **SetIgorMenuItem**, as described in Chapter 8, **Adding Menus and Menu Items**.

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. You can use the SAVE_WINDOW message instead.
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 164 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 164 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 164 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 164 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 164 for further information.

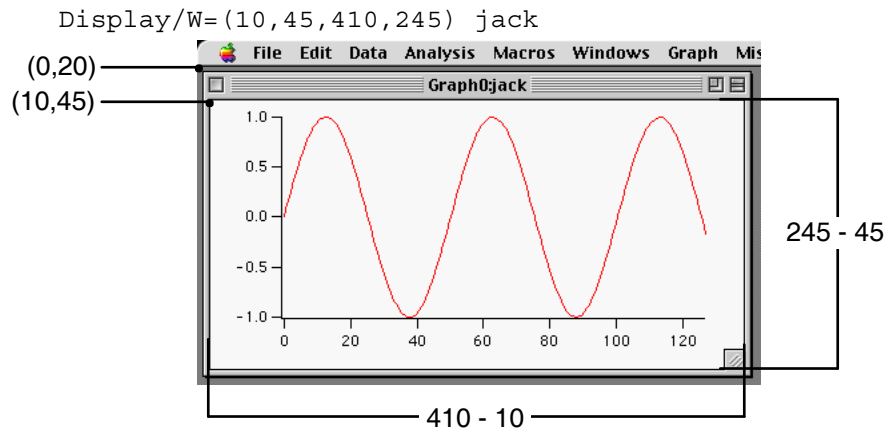
Recursion Problems

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 64.

Igor Window Coordinates

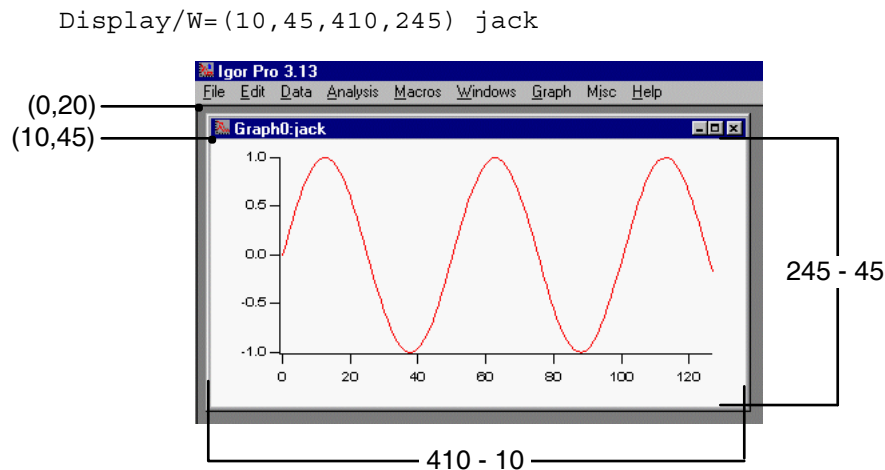
This section describes how Igor stores window sizes and positions in a platform-independent way. You need to know this if you are writing an XOP that creates a window on both Macintosh and Windows and if you want your window's size and position to be roughly maintained when you take an experiment created on one platform and open it on the other.

On the Macintosh, windows are sized and positioned using the `SizeWindow` and `MoveWindow` Carbon 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 Carbon API, 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



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 XP 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. On Windows 7 go to Control Panel→Appearance and Personalization→Display and click the "Set custom text size (DPI)" link.

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



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 **GetXOPWindowPositionAndState** 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 **SetXOPWindowPositionAndState** 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 “Gizmo” 3D 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.

Activating, Showing and Hiding Windows

Igor Pro 6 added features intended to make it easier to deal with large numbers of windows. These features, accessible through the Windows menu, include mass showing of windows, mass hiding of windows, and a Recent Windows submenu for retrieving recently activated windows. If you are creating a sophisticated XOP you may want to participate in these features. If so, here is what you must do. For an example, see the WindowXOP1 sample XOP.

When you receive the XOP_HIDE_WINDOW message from Igor, hide the window specified in the message. Use **HideXOPWindow**, not **HideAndDeactivateXOPWindow**, as you want to hide the window without sending it to the bottom of the desktop. You will receive the XOP_HIDE_WINDOW message if the user shift-clicks the close box on Macintosh, presses Cmd-Shift-W (Macintosh) or Ctrl-Shift-W (Windows) or if the user executes a DoWindow/HIDE=1 command on your window.

When you receive an XOP_SHOW_WINDOW message from Igor, show the window specified in the message. Use **ShowXOPWindow**, not **ShowAndActivateXOPWindow**, as you want to show the window without bringing it to the top of the desktop. You will receive the XOP_SHOW_WINDOW message if the user executes a DoWindow/HIDE=0 command on your window.

By convention, shift-clicking a window's close box hides the window rather than killing it. On Macintosh Igor detects this and sends you an XOP_HIDE_WINDOW message instead of a CLOSE message. However

on Windows you must detect the shift-click in your WM_CLOSE message and proceed accordingly, like this:

```
case WM_CLOSE:                // User clicked the close box.
    if ((GetAsyncKeyState(VK_SHIFT) & 0x8000) != 0) { // Shift key pressed?
        // Shift-click in close box hides window.
        HideXOPWindow(hwnd);
        TellIgorWindowStatus(hwnd, WINDOW_STATUS_DID_HIDE, 0);
        passMessageToDefProc = 0;
        break;
    }

    // Do normal close processing here.
```

When your window is hidden, shown, activated, deactivated, is about to be killed or is killed, call **TellIgorWindowStatus** using the appropriate one of these calls:

```
TellIgorWindowStatus(windowRef, WINDOW_STATUS_DID_HIDE, 0);
TellIgorWindowStatus(windowRef, WINDOW_STATUS_DID_SHOW, 0);
TellIgorWindowStatus(windowRef, WINDOW_STATUS_ACTIVATED, 0);
TellIgorWindowStatus(windowRef, WINDOW_STATUS_DEACTIVATED, 0);
TellIgorWindowStatus(windowRef, WINDOW_STATUS_ABOUT_TO_KILL, 0);
TellIgorWindowStatus(windowRef, WINDOW_STATUS_DID_KILL, 0);
```

NOTE: If you call this routine with any status, you must also call it when your window is about to be killed so Igor can remove it from various internal lists. Failure to do so may cause a crash.

WINDOW_STATUS_DID_KILL was added in Igor Pro 6.04 but you can pass that value to any version.

Chapter 10

Multithreading

Multithreading With XOPs

Multithreading allows a program to do more than one thing at a time. On a single-processor machine this works by giving a slice of time to each thread, giving the illusion of simultaneity. On a multiprocessor machine, multiple threads can execute at the same time on different processors, giving real simultaneity.

Multithreading is useful for two purposes:

- To allow a task to run without blocking other tasks
For example, you can perform data acquisition in a thread while you analyze previously-acquired data in the main thread. The main thread will not block the data acquisition thread.
- To achieve greater throughput by utilizing multiple processors
For example, you can run an analysis procedure on multiple data sets at one time using a thread for each data set.

Multithreading brings complications to programming. New rules must be followed to coordinate threads and keep them from interfering with one another. Programs that follow these rules are said to be "thread-safe".

Thread-safe programming is a vast topic which this manual will not cover except as it relates specifically to XOPs. We will assume that you understand the basics of thread-safe programming. If not, now would be a good time to familiarize yourself with it. A web search for "thread-safe programming" will get you started.

Multithreading is recommended only for experienced programmers. Whether you are experienced or not, you should verify that your XOP works correctly from the main thread before attempting multithreading.

Thread-safe External Operations and External Functions

For an external operation or function to be callable from a thread it must first be marked as thread-safe.

You mark an external operation as thread-safe using the XOPC resource that declares your operation and using the RegisterOperation callback that registers your operation with Igor. This is described under **Thread-Safe External Operations** on page 93.

You mark an external function as thread-safe using the XOPF resource that declares your function. This is described under **Thread-Safe External Functions** on page 114.

Once your external operation or external function is properly marked as thread-safe, Igor will allow you to call it from an Igor thread-safe user-defined function or to use it in a multithreaded computation (e.g., in a multithreaded curve fit). But marking it as thread-safe does not make it thread-safe - you must also write your operation or function in a thread-safe way. That is the subject of the following sections.

Three Kinds of Threads

We need to make a distinction between three kinds of threads:

- The main thread
- Igor preemptive threads
- Your own private internal preemptive threads that you create in your XOP

The *main thread* is created by the operating system when Igor is launched. Igor's outer loop and all user-interface activity run in the main thread. User-defined functions run in the main thread unless you explicitly create additional threads using Igor's ThreadGroupCreate and ThreadStart operations.

Igor preemptive threads are created by multithreaded Igor operations like FuncFit, when the MultiThread keyword is used to introduce a waveform assignment statement, and when an Igor user-defined function calls the ThreadGroupCreate operation.

You can use your own *private internal preemptive threads* in your XOP so long as Igor knows nothing about them. This means that you can not do a callback to Igor from one of your private internal threads.

NOTE: You can not do a callback to Igor from a private internal thread. You can do callbacks only from Igor's main thread and from Igor preemptive threads if the callback is designated as Igor-thread-safe.

Two Kinds of Igor Preemptive Threads

Igor preemptive threads can be classified as:

- Igor computation threads
- Igor user-defined function threads

An *Igor computation thread* is a thread created by a multithreaded Igor operation such as FuncFit or Optimize or by a multithreaded waveform assignment statement using the MultiThread keyword.

An *Igor user-defined function thread* is created when a procedure calls the ThreadGroupCreate operation. The newly-created thread sleeps until you call the ThreadStart operation. ThreadStart starts a user-defined function, called a "worker function", running in an Igor preemptive thread created by ThreadGroupCreate.

The following sections focus on creating thread-safe external operations and functions to be called from Igor user-defined function threads but the considerations discussed also apply to Igor computation threads.

Igor Messages and Threads

Igor can call your XOP in three ways:

- Igor sends a message to your XOP by calling its XOPEntry routine
- Igor calls your XOP's external operation from some thread
- Igor calls your XOP's direct external function from some thread

When we say that Igor sends a message to your XOP, we are speaking of the first of these three events. The mechanism by which Igor sends a message to an XOP is not thread-safe. Consequently, Igor will send a message to your XOP *from the main thread only*.

Writing Thread-safe Code

The topic of writing thread-safe code is vast and complex. In general, data that may be written by more than one thread, or written by one thread and read by another, requires special handling to prevent threads from interfering with each other. Your XOP's global variables are accessible by any thread and therefore create potential problems in a multithreaded environment. Globals that you set once at startup in the main thread

and only read from preemptive threads are OK. Globals that you read and write from the main thread and/or preemptive threads are problematic.

If your external operation or function uses only data (waves, numbers, strings, structures) passed to it through parameters and its own local variables, and does not use any read/write global variables, then your code will be thread-safe and you don't need to know any more about it. For example, a curve fitting function typically takes a parameter wave and an X value as an input and computes a numeric result by simply doing number crunching on the parameters. Such a function is thread-safe.

On the other hand, an external operation or function that stores state information in global variables is not thread-safe. It can be made thread-safe using multithreaded programming techniques such as mutexes and semaphores. This is for advanced programmers only who have studied the details of thread-safe programming and even then is tricky. You can greatly simplify matters by eliminating global variables as much as possible. Eliminating them altogether is ideal.

Igor Preemptive Thread Data Environment

When Igor creates a new preemptive thread it also creates a new data hierarchy for use by that thread and that thread only.

This means that each Igor preemptive thread has its own root data folder possibly containing waves and variables and sub-data folders. This simplifies thread-safe programming because you can modify waves and Igor variables in your data hierarchy without worrying that other threads are also reading or writing them.

When you do a callback such as `FetchWaveFromDataFolder`, which returns a handle to a named wave in a specific data folder, you are looking for a wave in your thread's data hierarchy, not in the main Igor data hierarchy. When you call `MDMakeWave`, you are making a wave in your thread's data hierarchy, not in the main Igor data hierarchy. The same is true for Igor global variables.

Waves Passed to Igor Threads as Parameters

In addition to data in the Igor thread's data hierarchy, an Igor preemptive thread can also access waves from the data hierarchy of the calling Igor thread (typically the main thread) that are passed as parameters to the thread's worker function. In the case of an Igor user-defined function thread, the worker function is the function specified in the `ThreadStart` command.

For example, here is a thread worker function that passes a wave from the calling thread to an external operation:

```
ThreadSafe Function ThreadWorker(threadInputWave)
    Wave threadInputWave    // Lives in calling thread's data hierarchy

    ExternalOperation threadInputWave
End
```

There are severe limits on what you are allowed to do with a thread parameter wave. It is the responsibility of the Igor user-function programmer to abide by these limits since the XOP programmer has no way to know if a given wave is a thread parameter wave.

Here is what you *can* do with a thread parameter wave:

- If the wave is numeric, you can read and/or write its data in the calling thread and in the receiving thread
- If the wave is text, you can read its data in the calling thread and in the receiving thread

If you write the wave's data in both threads, the contents of the wave will be unpredictable so you should not do that.

If you read the wave's data in one thread while the other thread is writing it, you may read some spurious data. This would occur if the reading thread does a read while a write is partially complete. If you are just displaying the wave in a graph or table, a transitory spurious read is not fatal. However, if your application can not tolerate a transitory spurious read, you must make some arrangement to coordinate the reading and writing threads.

Here is what you *can not* do with a thread parameter wave:

- You can not redimension the wave or change other wave properties
- You can not kill the wave

Igor returns an error if you attempt these forbidden operations on a thread parameter wave.

In addition:

- If the wave is text, you can not write to it

Igor returns an error if you attempt to write to a thread parameter text wave using **MDSetTextWavePointValue**.

Igor Pro 6.23 or later returns an error if you attempt to write to a thread parameter text wave using **SetTextWaveData**. Prior to Igor Pro 6.23, **SetTextWaveData** did not return an error in this circumstance. This could cause a crash if another thread was simultaneously reading the text wave.

The restrictions on a thread wave parameter end when the receiving thread's worker function returns.

Categories of XOPSupport Routines

XOPSupport routines can be categorized as either callback routines (e.g., **MDMakeWave**), which call back to Igor, and utility routines (e.g., **GetCStringFromHandle**), which do not call back to Igor.

In the following discussion, "callback routine" means an XOP Toolkit routine that calls back to Igor, "utility routine" means an XOP Toolkit routine that does not call back to Igor and "XOPSupport routine" means any XOP Toolkit routine (a callback or a utility).

Igor-Thread-Safe Callbacks

The thread-safety of each routine is listed in Chapter 15, **XOPSupport Routines**. Many of the routines are described as "Igor-thread-safe". This means that you can call them from Igor's main thread and from Igor preemptive threads when running with Igor Pro 6.20 or later. But you can not call such routines from your own private internal threads.

Thread-Safety of Callbacks from an Igor Preemptive Thread

In the past XOP callbacks to Igor were not thread-safe because the mechanism by which parameters were passed from the XOP to Igor was not suitable for multithreading.

The **old** rule was:

An XOP that is called from a preemptive thread can not do any callbacks to Igor.

Although you could create a thread-safe external operation or external function which could be called from an Igor thread, your operation or function was forbidden from doing callbacks to Igor. This represented a significant impediment for some XOP applications.

With XOP Toolkit 6 in combination with Igor Pro 6.20 or later, a new thread-safe mechanism is used for passing parameters to Igor. If an XOP is compiled with XOP Toolkit 6 and is running with Igor Pro 6.20 or later, the XOP library automatically uses the thread-safe mechanism.

The **new** rule is:

An XOP that is called from an Igor preemptive thread can do callbacks to Igor if the XOP was built using XOP Toolkit 6 and is running with Igor Pro 6.20 or later and if the callback is designated as Igor-thread-safe.

Chapter 15, **XOPSupport Routines** tells you which XOPSupport routines are Igor-thread-safe and which are not.

If you want to create a thread-safe external operation or external function that does callbacks to Igor, you must make sure that you are running with Igor Pro 6.20 or later. You can do this using the normal version check in your XOPMain function:

```
HOST_IMPORT int
XOPMain(IORecHandle ioRecHandle)
{
    XOPInit(ioRecHandle);    // Do standard XOP initialization.

    /* This XOP adds thread-safe external operations or functions that do
       callbacks to Igor from a thread. This requires Igor 6.20 or later.
    */
    if (igorVersion < 620) {
        SetXOPResult(IGOR_OBSOLETE);
        return EXIT_FAILURE;
    }

    . . .
```

If your XOP must run with versions of Igor prior to 6.20, more stringent rules apply as explained in the next section.

Thread-Safety With Older Versions of Igor

If your XOP is running with versions of Igor prior to 6.20 and you want to call your external operation or function from an Igor thread, you must follow the rule in effect before Igor 6.20 and XOP Toolkit 6.

The rule is:

An XOP that is called from a preemptive thread can not do any callbacks to Igor.

The reason for this is that the XOP callback mechanism used before Igor Pro 6.20 and XOP Toolkit 6 is not thread-safe. This restriction greatly limits what you can do from a thread-safe external operation or external function. However, it does not prevent you from writing some classes of external functions as thread-safe. For example, a pure number-crunching function, such as a curve-fitting function, can be written in a thread-safe fashion.

The following wave access functions do not do callbacks when running with Igor Pro 6:

```
WaveType, WavePoints, WaveData, WaveModDate, WaveModState, WaveModCount
MDGetWaveDimensions, MDGetWaveScaling, MDGetWaveUnits, MDSetWaveUnits
MDGetDimensionLabel, MDSetDimensionLabel, MDAccessNumericWaveData
MDGetNumericWavePointValue, MDSetNumericWavePointValue
MDGetDPDataFromNumericWave, MDStoreDPDataInNumericWave
```

All other wave access functions do callbacks to Igor and thus can not be called from preemptive threads when running with older versions of Igor. For example, the **WaveHandleModified** routine is a callback and so is off limits.

The **SetOperationNumVar**, **SetOperationStrVar** and **SetOperationStrVar2** routines, which are used to return information from external operations, are also callbacks. This makes it impractical to write a thread-safe external operation that runs with older versions of Igor.

Most other XOPSupport routines also do callbacks to Igor and thus can not be used from thread-safe code with older versions of Igor.

In addition to not calling any callbacks from a thread, you must also not call any XOPSupport utility routines that are marked as not thread-safe in Chapter 15, **XOPSupport Routines**.

As you can see, what you can do in a thread is severely restricted when running with older versions of Igor. Rather than coding to these restrictions, we recommend that you freeze the old version of your XOP for use by people running old versions of Igor and start a new version of your XOP that requires Igor Pro 6.20 or later. Then you can use the relaxed rules described in the next section in the new version.

Thread-Safety With Igor Pro 6.20 or Later

Even when you have built your XOP using XOP Toolkit 6 and you are running with Igor Pro 6.20 or later, not all XOPSupport routines are thread-safe. This is because some of the routines run code that is not written to be thread-safe. For example, the **XOPCommand** callback, which executes a command in Igor's command line, is not thread-safe.

In XOP Toolkit 6 each XOPSupport routine that is not thread-safe checks to see if it is running in a thread other than the main thread. If so it:

- Displays a dialog alerting you to the fact that you are calling a thread-unsafe routine from a thread
- Returns a result that indicates that an error occurred, if the routine return value is not void

If this happens, your XOP has a bug that you need to correct.

Routines that return an error code return `NOT_IN_THREADSafe` in this situation. Routines that return a handle, such as a wave handle or menu handle, return `NULL`.

It is important that you check the return value from every routine that returns a value and that you handle abnormal returns correctly. For example, if you inadvertently call a non-thread-safe routine from a thread and that routine normally returns a wave handle, it will return `NULL`. If you detect and deal with this correctly (e.g., by returning a `NOWAV` error to Igor), you will eventually get a reasonable error message. If you do not detect it, you will use the handle and your XOP will crash. Debugging an error message is much easier than debugging a crash, so make sure to test and correctly handle all return values.

XOPSupport routines that deal with user interface - including menus, dialogs and windows - are not thread-safe.

In general, XOPSupport routines that read or write data folders, waves and variables are thread-safe when you are running with Igor Pro 6.20 or later.

You Can Not Do Callbacks From Private Threads

A private thread is a thread that your XOP creates as opposed to a thread that Igor creates. Only very advanced XOPs will need to use private threads.

You can not do any callbacks from private threads.

The rest of this section explains the reason for this restriction.

Each Igor thread has its own data hierarchy consisting of a root data folder, child data folders, and waves and variables stored in those data folders. Normally each thread accesses its own data hierarchy only. (The exception is that a thread can access waves passed to it as parameters from another thread - see **Waves Passed to Igor Threads as Parameters** on page 169.)

This arrangement means that thread-safe user-defined functions, external operations and external functions do not need to worry about coordinating data access with other threads. They can access waves and variables in their own data hierarchy without the use of mutexes or other thread-safety techniques. This greatly simplifies Igor programming with threads.

Internally Igor maintains a data structure for each Igor thread that points to the thread's data hierarchy. To guarantee that a given thread accesses its data hierarchy only, Igor internally passes a pointer to this structure to any internal routine that needs to access data. For the purposes of this discussion, we will call this pointer an “Igor thread data pointer” or an ITDP for short. Just to be clear, an ITDP is a conceptual term for a pointer to a structure used internally by Igor to reference an Igor thread's data hierarchy.

When you do a callback from your XOP to Igor, Igor may need to access the current thread's data hierarchy and therefore needs an ITDP for the current thread. Igor obtains this from thread-local storage in which it stored the thread's ITDP when the thread was created. If you do a callback from an Igor thread, either the main thread or an Igor preemptive thread, Igor gets a valid ITDP from thread-local storage.

When you do a callback, Igor *assumes* that you are calling it from an Igor thread. If you do a callback from your *private* thread, Igor will attempt to get an ITDP and will get garbage. If the callback uses the ITDP, Igor will crash.

Another problem with doing callbacks from private threads is that there is no way to prevent a private thread and an Igor thread from changing Igor data (waves, variables and data folders) simultaneously.

Consequently, it is up to you to make sure that you refrain from doing callbacks to Igor from any private threads you create.

Determining if You are Running in a Preemptive Thread

Sometimes you might want to know if you are running in a preemptive thread so that you can do something one way if called from a thread or another way if called from the main thread. You can use the **RunningInMainThread** utility routine for this purpose.

For example, you may create an external operation that is thread-safe when running with Igor Pro 6.20 or later but not thread-safe when running with earlier versions of Igor. You would include the following test in your code:

```
if (igorVersion < 620) {
    if (!RunningInMainThread())
        return NOT_IN_THREADSafe;
}
```

NOT_IN_THREADSafe is a built-in Igor error code.

If you have code that should never run in a preemptive thread, you can use **CheckRunningInMainThread** to inform yourself that, contrary to your expectations, it is being called from a thread:

```
if (!CheckRunningInMainThread("SomeRoutineName"))
    return NOT_IN_THREADSafe;
```

Replace “SomeRoutineName” with the name of your routine. This displays a dialog informing you of the error.

CheckRunningInMainThread should be used only to inform yourself that your thread-safety checks have failed to do the job and that code that is never supposed to execute from a thread is doing so. The XOP Toolkit uses it to inform you that you are calling a non-thread-safe XOPSupport routine from a thread.

If **CheckRunningInMainThread** returns 0 you know you are running in a preemptive thread but it does not tell you if you are running in an Igor preemptive thread or in your own private internal thread. If you use private threads, it is up to you to keep track of this distinction.

Strategies For Threaded Data Acquisition

In this section we assume that you are running with Igor Pro 6.20 or later and want to acquire data in an Igor preemptive thread or in a private preemptive thread created by your XOP so that you can use Igor for other purposes while data is being acquired. This is recommended for experienced programmers only.

In order to display data in a graph or table or to pass it to non-thread-safe Igor operations, it needs to be in the data hierarchy of Igor's main thread. There are four methods to achieve this. In two of the methods, the data is acquired in an Igor preemptive thread. In the other two, it is acquired in a private thread that your XOP creates. The first method has two variations:

Method	Data Acquired In	Data Acquired To	Data Transfer to Main Thread
Method 1A (page 175)	Igor Thread	Igor thread's data hierarchy	The Igor thread worker function, running in the preemptive thread, sends data to Igor's thread output queue. An Igor named background task, running in the main thread, reads data from the thread output queue.
Method 1B (page 176)	Igor Thread	Igor thread's data hierarchy	The external operation or external function, running in the preemptive thread, sends data to Igor's thread output queue. The XOP, running in the main thread, transfers data from the output queue to the main thread's data hierarchy when Igor sends the IDLE message to the XOP or when an external operation or external function created for that purpose is called. This method requires Igor Pro 6.23 or later.
Method 2 (page 177)	Private Thread	XOP private data buffer	The XOP, running in the main thread, transfers data from its private buffer to the main thread's data hierarchy when Igor sends the IDLE message to the XOP or when an external operation or external function created for that purpose is called.
Method 3 (page 178)	Igor Thread	Main thread numeric waves	No transfer required
Method 4 (page 178)	Private Thread	Main thread numeric waves	No transfer required

The first two methods are clean but require more work on your part. The last two methods are quick and dirty. The last method is not recommended for XOPs that need to be bulletproof.

If you are comfortable creating your own threads and coordinating between threads then method 2 is preferred. Otherwise method 1A or 1B are preferred. Method 3 is the easiest to implement.

The following sections describe each of the methods. They assume that you are familiar with thread-safe programming in Igor itself. If not, you can come up to speed by:

- Reading the "ThreadSafe Functions and Multitasking" section of Igor's help. This is also available in the "Advanced Programming" chapter of the Igor Pro manual.
- Examining the "Slow Data Acq" example experiment. You can open it by choosing File→Example Experiments→Programming→Slow Data Acq.

The Slow Data Acq example experiment can be used as a starting point for your own Igor data acquisition project.

The following sections also assume that you have created an external operation to acquire data but an external function would also work.

Method 1A: Acquiring Data in an Igor Thread

In this variant of method 1, data is acquired in an Igor preemptive thread. It is transferred to Igor's thread output queue by the thread worker function. It is transferred from the thread output queue to the main thread's data hierarchy by a named background task.

Here is a schematic view of this method:

Igor Main Thread	Igor Preemptive Thread
A user-defined function creates a thread group (ThreadGroupCreate) and starts a thread worker function (ThreadStart)	
The user-defined function starts a named background task (CtrlNamedBackground) which periodically looks for data from the Igor preemptive thread (ThreadGroupGetDFR)	The thread worker function creates a data folder in the thread's private data hierarchy, sets it as the thread's current data folder, and calls the external operation
The user-defined function returns and the main thread is now free for interactive use	The external operation acquires data into the thread's current data folder, creating waves, variables and sub-data folders
	The external operation returns to the thread worker function
	The thread worker function sends data to the thread output queue (ThreadGroupPutDF)
The named background task receives data from the thread output queue and transfers it to the main thread (ThreadGroupGetDFR)	If more data is to be acquired, the thread worker function repeats the process
	The thread worker function returns and the Igor preemptive thread stops

Igor creates a separate data hierarchy for each Igor thread. This means that your thread worker function and your external operation can create data folders, waves and variables and can read and write waves without worrying about interference from other threads.

The tricky part is getting the data from the Igor preemptive thread into Igor's main thread so you can display it. Prior to Igor Pro 6.23, this could be done only from within Igor itself. Furthermore, it requires the participation of two threads: your preemptive thread and Igor's main thread.

Your thread worker function starts data acquisition by creating a data folder into which data will be acquired and then calling your external operation which runs until it has acquired the data. When your external operation returns, your thread worker function calls Igor's ThreadGroupPutDF operation to transfer the data folder to Igor's thread output queue. Your thread worker function can then call your external operation again or it can simply return if data acquisition is finished.

In the mean time, a user-defined function running in Igor's main thread calls ThreadGroupGetDFR to retrieve the data from Igor's thread output queue. To avoid monopolizing the main thread, this user-defined function must be a named background task. See "Background Tasks" in the Igor help or in the "Advanced Programming" chapter of the Igor manual for details.

Chapter 10 — Multithreading

After the data has been transferred, the background task function can process and/or display it by directly calling Igor operations or other user-defined functions.

The Slow Data Acq sample experiment simulates this entire process. It uses a user-defined function rather than an external operation to do the data acquisition. Carefully read the documentation and code included in that experiment. You can then use it as a starting point for your own XOP-based threaded data acquisition.

Method 1B: Acquiring Data in an Igor Thread

This variant of method 1 is the same as method 1A except that data is transferred to and from the Igor preemptive thread group's output queue by the XOP instead of by user-defined functions. This requires Igor Pro 6.23 or later because the **ThreadGroupPutDF** and **ThreadGroupGetDF** callbacks were added in that Igor version. If it is acceptable to require Igor Pro 6.23, this variant simplifies matters by eliminating the named background task, assuming there are not other reasons to use a background task.

If you use this variant, check the Igor version as shown under **Thread-Safety of Callbacks from an Igor Preemptive Thread** on page 170 but use 623 instead of 620.

In this variant of method 1, data is acquired in an Igor preemptive thread. It is transferred to Igor's thread output queue by the external operation running in the preemptive thread. It is transferred from the thread output queue to the main thread's data hierarchy by the external operation running in the main thread.

Here is a schematic view of this method:

Igor Main Thread	Igor Preemptive Thread
A user-defined function creates a thread group (ThreadGroupCreate) and starts a thread worker function (ThreadStart)	
The user-defined function returns and the main thread is now free for interactive use	The thread worker function calls the external operation to start data acquisition
	The external operation creates a data folder in the thread's private data hierarchy, sets it as the thread's current data folder, and acquires data into it, creating waves, variables and sub-data folders
	The external operation calls the ThreadGroupPutDF callback to transfer the data folder to the thread group's output queue
	The external operation returns to the thread worker function
Igor sends the IDLE message to the XOP	
The XOP calls ThreadGroupGetDF to see if data from the preemptive thread is available and to transfer it to the main thread's data hierarchy	If more data is to be acquired, the thread worker function repeats the process
	The thread worker function returns and the Igor preemptive thread stops

Igor creates a separate data hierarchy for each Igor thread. This means that your thread worker function and your external operation can create data folders, waves and variables and can read and write waves without worrying about interference from other threads.

The tricky part is getting the data from the Igor preemptive thread into Igor's main thread so you can display it. This requires the participation of two threads: your preemptive thread and Igor's main thread.

Prior to Igor Pro 6.23, this could be done only from within Igor itself. Because the **ThreadGroupPutDF** and **ThreadGroupGetDF** callbacks were added in that Igor version, it can now be done entirely in the XOP.

Your thread worker function calls your external operation to start data acquisition. The external operation acquires the data, storing it in a data folder in the Igor thread's private data hierarchy. The external operation then calls the **ThreadGroupPutDF** callback to transfer the data folder to Igor's thread output queue.

In the mean time, Igor's sends IDLE message to your XOP from the main thread. Your XOP calls the **ThreadGroupGetDF** callback to transfer a data folder from Igor's thread output queue to the main thread's data hierarchy if one is available. Your XOP may then invoke Igor operations or a user-defined function via a callback such as **XOPCommand2** to display or further process the data.

Igor sends the IDLE message periodically when nothing else is going on - that is when no user-defined function or Igor operation is running in the main thread. You can force Igor to send an IDLE message to all XOPs by calling the **DoXOPIdle** operation from a user-defined function running in the main thread.

You can also create an external operation that causes your XOP to transfer data to the main thread, as if Igor had sent the IDLE message. You would call this external operation from the main thread only. This is more efficient than calling **DoXOPIdle** because **DoXOPIdle** sends IDLE messages to all XOPs.

Method 2: Acquiring Data in a Private Thread

In this method, data is acquired into your private buffer in a private thread that you create in your XOP. When your XOP receives an IDLE message from Igor, which occurs only in the main thread, it transfers data from the private buffer to Igor's main thread.

Here is a schematic view of this method:

Igor Main Thread	XOP Private Thread
A user-defined function calls an external operation to start data acquisition	
The external operation creates a private buffer into which data will be acquired and starts a private thread	The private thread acquires data into the private buffer
The user-defined function returns and the main thread is now free for interactive use	
Igor sends the IDLE message to the XOP	
If data is ready, the XOP transfers data from the private buffer to waves in the main thread	
If data acquisition is finished, the XOP stops the private thread	

Igor sends the IDLE message periodically when nothing else is going on - that is when no user-defined function or Igor operation is running in the main thread. You can force Igor to send an IDLE message to all XOPs by calling the **DoXOPIdle** operation from a user-defined function running in the main thread.

You can also create an external operation that causes your XOP to transfer data to the main thread, as if Igor had sent the IDLE message. You would call this external operation from the main thread only. This is more efficient than calling **DoXOPIdle** because **DoXOPIdle** sends IDLE messages to all XOPs.

Since your private buffer and any related control and status variables are shared by your private preemptive thread and your code running in the main thread, you need to use thread-safe programming techniques, such as mutexes, to coordinate thread activity.

Method 3: Acquiring Data in an Igor Thread Directly To Main Thread Waves

This method is simpler but has some significant limitations. You write an Igor user-defined function that creates a thread group and starts a thread worker function. You pass waves created in the main thread as parameters to your thread worker function. Your thread worker function calls your external operation which stores data in the waves.

Here is a schematic view of this method:

Igor Main Thread	Igor Preemptive Thread
A user-defined function creates waves to receive the acquired data	
The user-defined function creates a thread group (ThreadGroupCreate) and starts a thread worker function (ThreadStart), passing waves to the thread worker function	
The user-defined function returns and the main thread is now free for interactive use	The thread worker function calls the external operation, passing the waves as parameters
	The external operation acquires data into the waves and marks them as modified (WaveHandleModified)
	The external operation returns to the thread worker function
	The thread worker function returns and the Igor preemptive thread stops

Your thread worker function passes its parameter waves, which were created in the main thread, to your external operation. Your external operation stores data in the parameter waves and calls **WaveHandleModified** to tell Igor that the waves' data was changed. If the waves are displayed in a graph or table, Igor will update them in its main loop. You could call WaveHandleModified periodically to get periodic updates.

Waves passed as parameters from one thread to another are marked by Igor as in use by more than one thread and Igor returns an error if you attempt to change the waves, for example, by redimensioning them. However, you are allowed to change the data of numeric waves only. The restrictions on such waves are further discussed under **Waves Passed to Igor Threads as Parameters** on page 169.

The restrictions on a thread wave parameter end when the receiving thread terminates which is normally when the thread worker function returns.

Because you can not redimension thread parameter waves, you have to know how big to make them when data acquisition starts.

Method 4: Acquiring Data in a Private Thread Directly To Main Thread Waves

This method does not require any Igor preemptive threads but comes with even more restrictions than the preceding method. You call your external operation from Igor's main thread, passing waves to it as parameters. Your external operation obtains the addresses of the parameter waves' data by calling **WaveData** or **MDAccessNumericWaveData**. It then creates a private internal thread and passes the wave data addresses to it. Your external operation then returns, allowing Igor's main thread to continue. Your private internal thread writes directly to the waves using the technique described under **The Direct Access Method** on page 124.

Here is a schematic view of this method:

Igor Main Thread	XOP Private Thread
A user-defined function creates waves to receive the acquired data	
The user-defined function calls an external operation to start data acquisition, passing the waves to it	
The external operation starts a private thread and returns	The private thread acquires data into the waves
The user-defined function returns and the main thread is now free for interactive use	
The XOP receives IDLE messages from Igor and checks if data acquisition is finished	When data acquisition is finished, the private thread terminates
The XOP, detecting that data acquisition is finished, marks the waves as modified (WaveHandleModified)	

All of the restrictions discussed under **Waves Passed to Igor Threads as Parameters** on page 169 apply to this method. From a private thread, all you can do is read or write the data of numeric parameter waves. In addition, you can not do any callbacks to Igor from a private thread.

This last restriction means that you can not tell Igor that a wave is modified by calling **WaveHandleModified** from a private thread. You can do this when your XOP receives the IDLE message in the main thread or from an external operation running in the main thread. In order to know when to call WaveHandleModified, you will need to devise some coordination between your private thread and the main thread.

In addition, you must meet these requirements:

- To prevent a crash, the main thread must not kill, redimension or otherwise modify the parameter waves while the private thread is accessing them
- To prevent data corruption, the main thread must not write to the parameter waves while the private thread is accessing them
- To prevent a crash, if you receive the NEW or CLEANUP messages, signifying that waves are about to be killed, you must terminate your private threads

Because of the possibility of a crash, this method is not recommended for XOPs that must be bulletproof.

Finally, the caveats about spurious data in the preceding section apply to this method also.

Chapter 11

64-bit XOPs

Overview

XOP Toolkit 6 supports development of 32-bit Macintosh XOPs and 32-bit and 64-bit Windows XOPs.

A Windows-only 64-bit beta version of Igor is available for download. We refer to the 64-bit version as IGOR64. The 32-bit version is called IGOR32.

A 64-bit Macintosh Igor will be available with Igor7. You will need XOP Toolkit 7 to create a 64-bit Macintosh XOP and it will require Igor7 or later..

IGOR32 can theoretically address up to 4 GB of virtual memory. In practice that actual limit is somewhere between 2 GB and 4 GB depending on your hardware, OS, and application. Most users will never come anywhere close to this limit.

IGOR64 is of interest to Igor users who need to access very large amounts of data at one time. IGOR64 can theoretically address 2^{64} bytes of virtual memory, or roughly 17 billion GB. However, current 64-bit processor architectures and the Windows OS impose further limits. In 64-bit Windows operating systems each process can address 8 terabytes (about 8000 GB). So IGOR64 works in a virtual address space of 8 terabytes. In practice, you are unlikely to hit this limit.

You can find a discussion of operating system address space limits here:

[http://msdn.microsoft.com/en-us/library/aa366778\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa366778(VS.85).aspx)

The maximum practical amount of data that you can access is determined by how much physical RAM you have in your system, your application, and how much patience you have. Even if you have plenty of physical RAM, processing gigabytes of data takes a long time.

IGOR32 can use 32-bit XOPs only. IGOR64 can use 64-bit XOPs only. It is not possible to run a 32-bit XOP with a 64-bit Igor or vice versa. Consequently, if you have a 32-bit XOP that you want to use with IGOR64 then you will have to port your XOP to 64 bits. The following sections explain how to do that.

Igor requires that 64-bit XOP file names end with "64" (e.g., GBLoadWave64.xop). If the XOP name ends with a digit (e.g., VDT2.xop), append "-64" (e.g., VDT2-64.xop). If the XOP name does not end with 64, followed by the .xop extension, IGOR64 will not load it.

A 32-bit XOP is activated by putting a shortcut pointing to it in the "Igor Extensions" folder. A 64-bit XOP is activated by putting a shortcut pointing to it in the "Igor Extensions (64-bit)" folder.

To learn more about IGOR64, visit:

[http://www.wavemetrics.net/Updaters/ReadMe%20\(64-bit\).html](http://www.wavemetrics.net/Updaters/ReadMe%20(64-bit).html)

Using Visual C++ For 64-bit Development

To develop a 64-bit XOP you need Visual C++ 2008 (VC9) or later. The standard edition supports 64-bit development by default. You can also use Visual C++ 2010 (VC10) which as no standard edition - just a professional (more expensive) edition.

The express (free) editions of Visual C++ do not support 64-bit development by default. Microsoft web pages say that you can enable 64-bit development by installing additional software but this did not work when we tried it so we bought Visual C++ 2008 (VC9) standard edition.

Changing a Visual C++ Project for 64-bit Development

This section describes how we changed the GBLoadWave sample XOP project to add 64-bit configurations following Microsoft instructions at:

<http://msdn.microsoft.com/en-us/library/9yb4317s.aspx>

Visual C++ supports the idea of "configurations". A configuration is a collection of settings used to build a project for a particular purpose. Visual C++ projects typically have a debug configuration and a release configuration. The debug configuration holds settings suitable for creating a debuggable version of the project's executable while the release configuration holds settings that do not support debugging but provide better performance. You typically use the debug configuration during development and the release configuration to create the executable for distribution to users.

To support 64-bit development, Visual C++ adds the idea of "platforms". The *Win32* platform is the traditional 32-bit platform and the *x64* platform is the 64-bit platform. If you have developed XOPs in the past they were for the Win32 platform.

To compile a 64-bit version of the GBLoadWave XOP, we need to add the x64 platform to the project. This will give us a total of four configurations: Win32-Debug, Win32-Release, x64-Debug and x64-Release.

Here are the steps we used to add the x64 platform to GBLoadWave using VC9. VC10 works the same. You can use this procedure to extend your existing XOP project to 64-bits.

NOTE: These instructions assume that you have already updated your XOP project as described under **Visual C++ Project Changes For XOP Toolkit 6** on page 460.

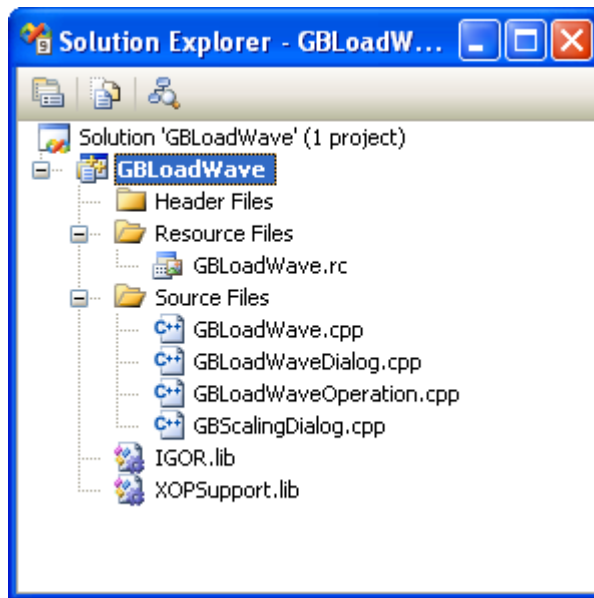
1-Visual C++.

Double-click the GBLoadWave.sln solution file in GBLoadWave\VC9.

Visual C++ starts up.

2-Visual C++.

In the Solution Explorer window, select the GBLoadWave icon.

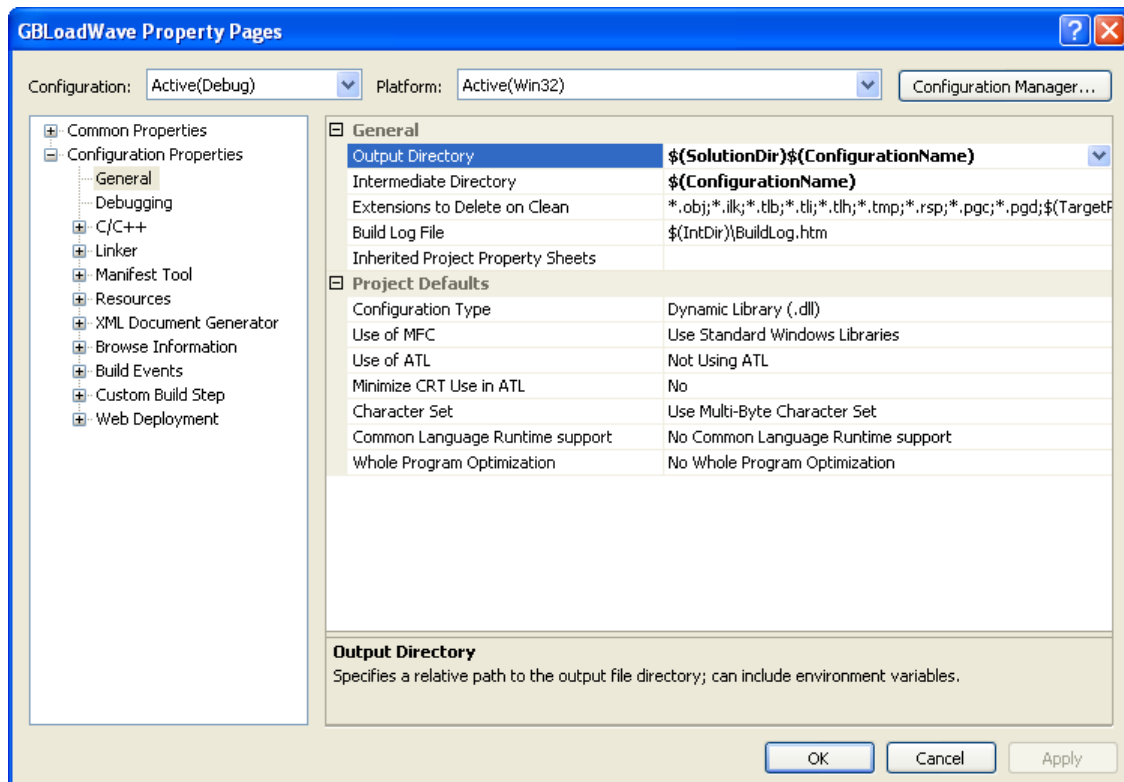


3-Visual C++.

Choose Project->Properties.

The GBLoadWave Properties window opens.

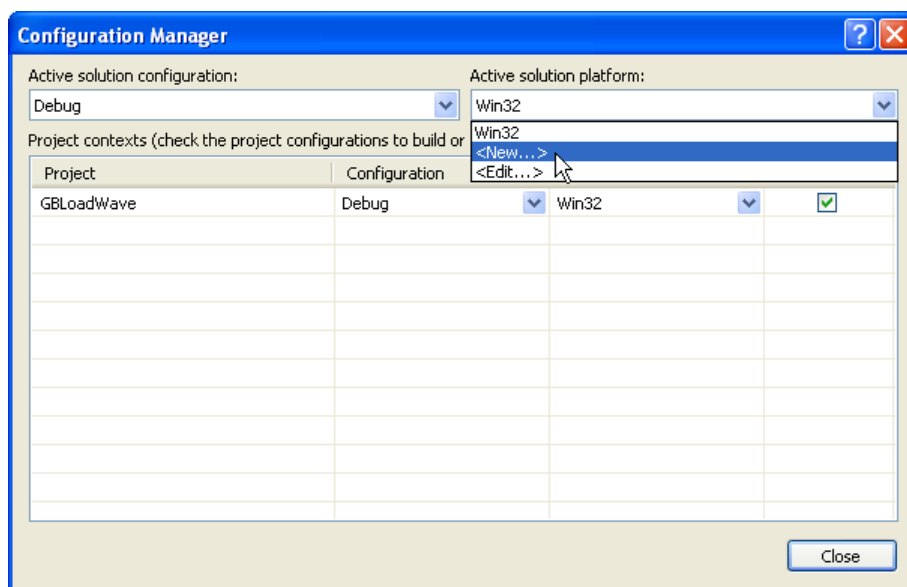
Open the Configuration Properties node of the tree view control.



4-Visual C++.

Click the Configuration Manager button.

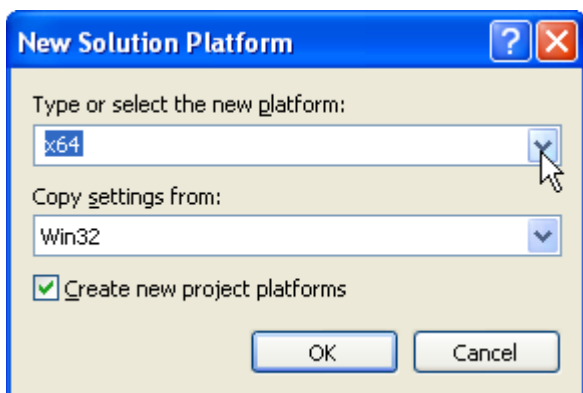
The Configuration Manager window opens.



5-Visual C++.

Click the menu under Active Solution Platform and choose New (as shown above).

The New Solution Platform window opens.



6-Visual C++.

Choose x64 from the top popup menu (as shown above).

Check the Create New Project Platforms checkbox.

If you do not see x64 in the top popup menu then your Visual C++ installation does not support 64-bit development. See **Using Visual C++ For 64-bit Development** on page 182 for further discussion.

The window should now look as shown above. Verify that Win32 is selected in the second popup menu and the Create New Project Platforms is checked.

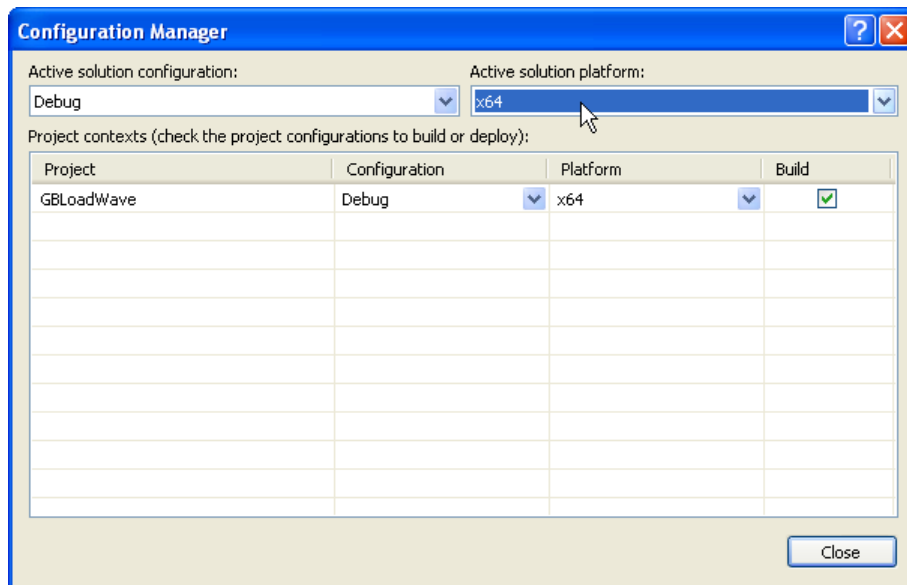
7-Visual C++.

Click OK in the New Solution Platform window.

The x64 platform has been added to your project. You now have four configurations: Win32-Debug, Win32-Release, x64-Debug and x64-Release.

Visual C++ created the x64 configurations by copying the settings from the Win32 configurations and then changing those settings that are 64-bit specific.

You are back in the Configuration Manager window which now looks like this:



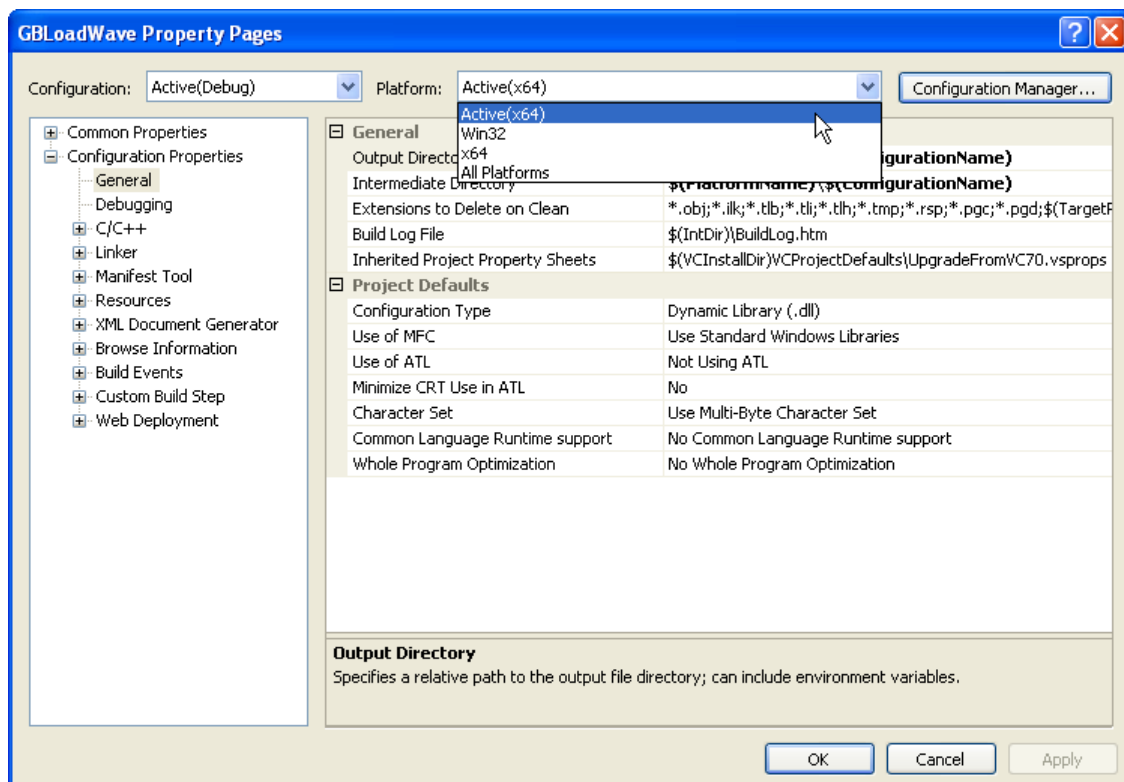
x64 is selected in the Active Solution Platform menu.

Using the two menus, you could display each of the four configurations and use the checkbox in the Build column to determine if a particular configuration is to be built by the project. This is not necessary as all configurations are set to be built at this point.

8-Visual C++.

Click Close in the Configuration Manager window.

You are now back in the GBLoadWave Properties window and you now have both Win32 and x64 platforms in the project:



Using the two menus at the top of the window, you can view and edit the settings for each of the four configurations in the project.

We will now fix some settings that were not set correctly when Visual C++ added the x64 platform configurations. Please follow these steps carefully.

9-Visual C++.

Select All Configurations from the Configuration popup menu.

Select x64 from the Platform menu.

Select Debugging from the Configuration Properties node of the tree view control.

Enter the following setting:

Debugging

Command: <path to your Igor64.exe>

where <path to your Igor.exe> is typically

C:\Program Files\WaveMetrics\Igor Pro Folder\Igor64.exe

or

C:\Program Files (x86)\WaveMetrics\Igor Pro Folder\Igor64.exe

You can set it using the Browse item in the pop-down menu for the Command item.

Click the Apply button.

10-Visual C++.

Select All Configurations from the Configuration menu.

Select x64 from the Platform menu.

Open the Linker icon in the tree control and select the General icon.

Change this setting as indicated:

Linker

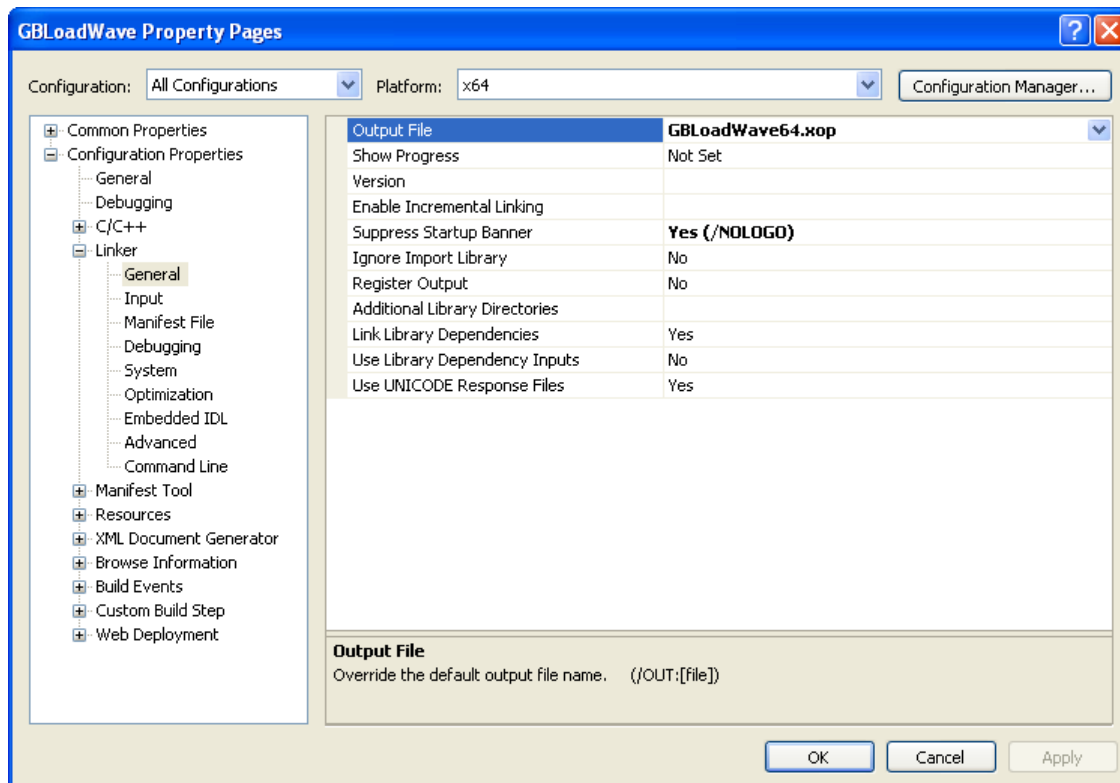
General

Output File: GBLoadWave64.xop

Click the Apply button.

Igor Pro requires that 64-bit XOP file names end with "64" so you can readily see that they are 64-bit XOPs. If the XOP name ends with a digit (e.g., VDT2.xop), append "-64" (e.g., VDT2-64.xop).

The Properties window now looks like this.



11-Visual C++.

Under the Linker icon in the tree control and select the Command Line icon.

Delete all text from the Additional Options section at the bottom of the window.

Click the Apply button.

Some XOP projects may have "/MACHINE:I386" in the Additional Options section. This will cause the linker to emit a LNK2112 error when compiling a 64-bit executable so this step removes it from the x64 configurations.

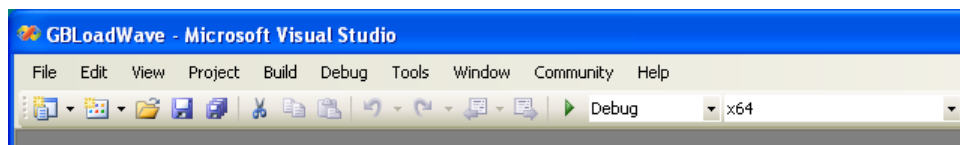
12-Visual C++.

Click OK.

The Properties window closes.

The tool bar under the Visual C++ menu bar allows you to choose which configuration is to be built. From the left menu you can choose Debug or Release. From the right menu you can choose Win32 or x64.

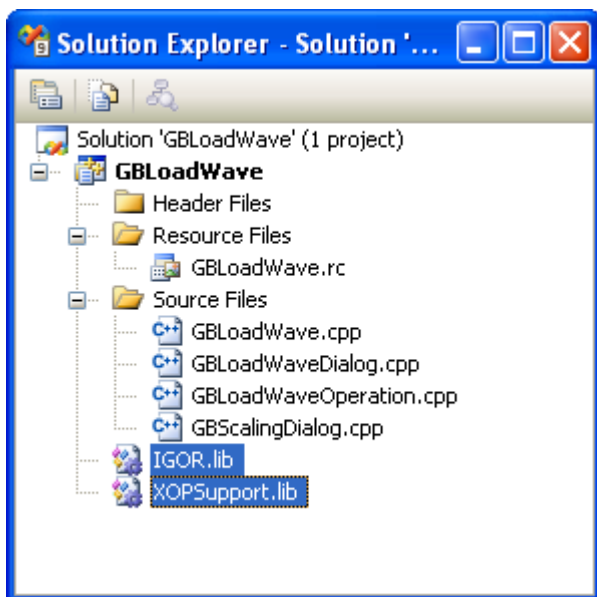
Choose Debug as the configuration and x64 as the platform:



The IGOR.lib and XOPSupport.lib files in the project are 32-bit libraries. We will now set them to be used only when building the 32-bit version of the XOP.

13-Visual C++.

In the Solution Explorer window, select the IGOR.lib and XOPSupport.lib icons.



14-Visual C++.

Right-click one of the selected icons and choose Properties.

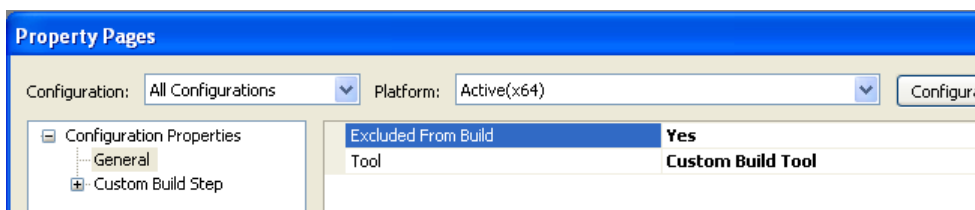
The Properties window opens.

15-Visual C++.

Select All Configurations from the Configuration menu.

Select x64 from the Platform menu.

Set Excluded From Build to Yes in the General category.



Click the Apply button.

Click OK.

Now IGOR.lib and XOPSupport.lib will be used for the Win32 configurations but not for the x64 configurations.

In the next two steps we will add IGOR64.lib and XOPSupport64.lib to the x64 project.

16-Visual C++.

In the Solution Explorer window, right-click the GBLoadWave icon and choose Add->Existing Item.

The Open File dialog is displayed.

Navigate to the IgorXOPs6\XOPSupport directory.

Choose All Files from the Files of Type menu at the bottom of the Open File dialog.

Select the IGOR64.lib icon and click Add.

If you get a dialog asking about "custom build rules", click No.

17-Visual C++.

In the Solution Explorer window, right-click the GBLoadWave icon and choose Add->Existing Item.

The Open File dialog is displayed.

Navigate to the IgorXOPs6\XOPSupport\VC9 directory.

Choose All Files from the Files of Type menu at the bottom of the Open File dialog.

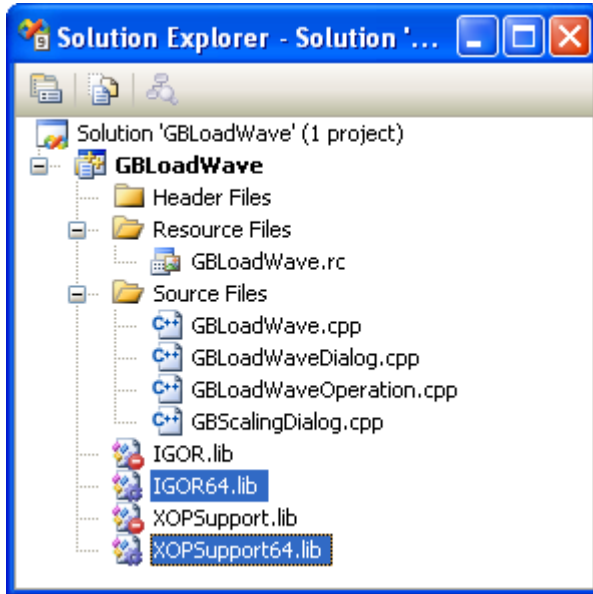
Select the XOPSupport64.lib icon and click Add.

If you get a dialog asking about "custom build rules", click No.

The IGOR64.lib and XOPSupport64.lib files in the project are 64-bit libraries. We will now set them to be used only when building the 64-bit version of the XOP.

18-Visual C++.

In the Solution Explorer window, select the IGOR64.lib and XOPSupport64.lib icons.



Make sure that only IGOR64.lib and XOPSupport64.lib are selected.

19-Visual C++.

Right-click one of the selected icons and choose Properties.

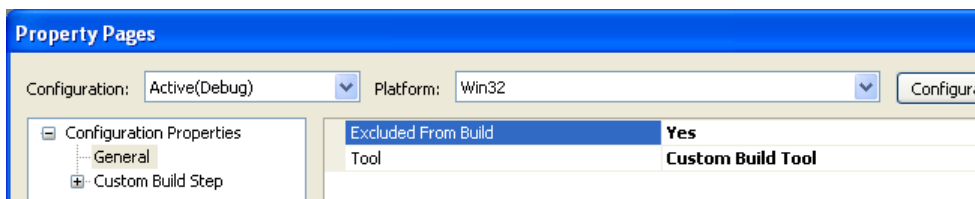
The Properties window opens.

20-Visual C++.

Select All Configurations from the Configuration menu.

Select Win32 from the Platform menu.

Set Excluded From Build to Yes in the General category.



Click the Apply button.

Click OK.

Now IGOR64.lib and XOPSupport64.lib will be used for the x64 configurations but not for the Win32 configurations.

21-Visual C++.

Choose File->Save All.

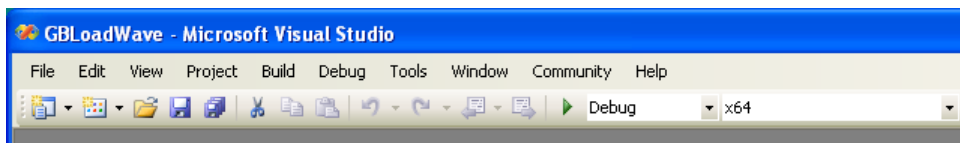
The project is now configured for 64-bit as well as 32-bit compilation.

Building the 64-bit XOP

We can now build the 64-bit XOP.

1-Visual C++.

Choose Debug and x64 from the Visual C++ menus in the tool bar below the menu bar.



2-Visual C++.

Choose Build Solution from the Visual C++ Build menu.

The 64-bit version of GBLoadWave, named GBLoadWave64.xop, is built.

It is likely that you will get compiler warnings when compiling for 64 bits that you did not get when compiling for 32 bits. This is discussed in the next section.

Activating a 64-bit XOP

You activate a 64-bit XOP by putting the XOP or a shortcut pointing to it in the “Igor Extensions (64-bit)” folder which you can find in your “Igor Pro User Files” folder. See **Activating XOPs Using the Igor Extensions Folder** on page 5 for details.

Data Types For 64-bit Compatibility

When compiling a 64-bit executable, a long on Windows is 32 bits while a long on Macintosh is 64 bits. Consequently the type long is problematic when compiling for 64 bits. Therefore in XOP Toolkit 6, all longs have been changed to ints or to the appropriate WaveMetrics data type.

Sometimes we need a value to be 32 bits when running a 32-bit XOP with IGOR32 but 64 bits when running a 64-bit XOP with IGOR64. Examples of such values are pointers, byte counts and wave point numbers.

In other cases we need a value to be 32 bits on both IGOR32 and IGOR64, for example, when reading or writing a field of a structure that is written to disk and read by both versions of Igor.

The following data types are defined in XOPSupport/WMTTypes.h which is #included as by XOPStandard-Headers.h. These data types are used to write code that works correctly with both 32-bit and 64-bit compilers on both Macintosh and Windows.

Types that are the same for 32-bit and 64-bit XOPs

These data types are the same in 32-bit and 64-bit XOPs.

Type	Description
SInt8	signed char (8 bits)
UInt8	unsigned char (8 bits)
SInt16	signed short (16 bits)

Type	Description
UInt16	unsigned char (16 bits)
SInt32	signed short (32 bits)
UInt32	unsigned char (32 bits)
TickCountInt	Tick count integer (32 bits) Used to hold a 32-bit unsigned tick count A tick is approximately 1/60th of a second
SInt64	signed 64-bit value
UInt64	unsigned 64-bit value

Types that are the different for 32-bit and 64-bit XOPs

These data types have different sizes in 32-bit and 64-bit XOPs.

Type	Description
PSInt	Pointer-sized integer 32 bits in IGOR32, 64 bits in IGOR64 Used for integers that may hold pointers and for other situations where an integer should hold 32 in IGOR32 and 64 bits in IGOR64
BCInt	Byte-count integer 32 bits in IGOR32, 64 bits in IGOR64 Used to hold a byte count that needs to be 32 bits in IGOR32 and 64 bits in IGOR64
IndexInt	Index integer 32 bits in IGOR32, 64 bits in IGOR64 Used to hold a value used to index a pointer Also used to hold wave point numbers
CountInt	Count integer 32 bits in IGOR32, 64 bits in IGOR64 Used to hold an item count that needs to be 32 bits in IGOR32 and 64 bits in IGOR64 such as the size of a wave dimension

Programming For 64-bit Compatibility

This section discusses some issues that you need to be aware of when writing a 64-bit compatible XOP.

Background Information

It is a good idea to familiarize yourself with 64-bit porting issues before writing 64-bit code.

Here is Microsoft's 64-bit porting guide:

[http://msdn.microsoft.com/en-us/library/bb427430\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/bb427430(VS.85).aspx)

Here is Apple's 64-bit porting guide. Although there is currently no 64-bit Macintosh Igor, Apple's guide is still a good source of background information:

<http://developer.apple.com/documentation/Darwin/Conceptual/64bitPorting/64bitPorting.pdf>

64-bit Compiler Warnings

It is likely that you will get compiler warnings when compiling for 64 bits that you did not get when compiling for 32 bits. In many cases, you will need to use a cast to stop the compiler warning. For example, this line:

```
int len = strlen(name);
```

will give you a compiler warning because `strlen` returns a 64-bit integer and we are assigning it to a 32-bit variable. You can fix it with a cast:

```
int len = (int)strlen(name);
```

This cast is OK because we know that the length of the name string will fit in 32 bits.

Another example:

```
int len = GetHandleSize(waveH);
```

In this case, the value being stored in `len` could exceed 32 bits so this warning should be fixed like this:

```
BCInt len = GetHandleSize(waveH);
```

`BCInt` is a data type defined by the XOP Toolkit to hold byte counts. It is a 32-bit value when compiling for 32 bits and a 64-bit value when compiling for 64 bits. You could also use the standard C `size_t` data type.

If you know that the handle size can not exceed 32 bits then it is OK to store it in an `int` and to cast the result from `GetHandleSize` to an `int`:

```
int len = (int)GetHandleSize(fileNameH);
```

The long data type

On Windows in 64 bits, a `long` is a 32-bit integer as it was in 32 bits. However, on Macintosh and on Unix systems, a `long` is 64 bits. Because of this dichotomy, if you want to write cross-platform 64-bit code, you must get rid of `long`s in your code.

If you are concerned about Windows only then it is not necessary to get rid of `long`s. However, if you are going to spend the time to make your code 64-bit clean, it makes sense to do the whole job at once by getting rid of `long`s even though it is not necessary for Macintosh.

Unfortunately there is no single data type to represent a 64-bit integer on both Macintosh and Windows. Xcode uses "long long" while Visual C++ uses "`__int64`".

To deal with this and other issues, XOP Toolkit 6 defines portable data types in `WMTypes.h`. `WMTypes.h` is `#included` by `XOPStandardHeaders.h`. `WMTypes.h` defines `SInt64` as a signed 64-bit integer regardless of platform. It also defines several other useful cross-platform data types - see **Data Types For 64-bit Compatibility** on page 190 for the complete list.

There are four situations with regard to `long`s:

- 32 bits is sufficient
Example: A general purpose flag or count that will always fit in 32 bits
Resolution: Change the `long` to an `int`
- 32 bits is always required
Example: A field in a structure that must always be exactly 32 bits
Resolution: Change the `long` to an `SInt32`
- 64 bits is always required
Example: A field in a structure that must always be exactly 64 bits

Resolution: Change the long to an SInt64

- 32 bits is required when running in 32 bits, 64 bits is required when running in 64 bits

Example: A pointer or a count or index that can be arbitrarily large

Resolution: Change the long to a BCInt, CountInt, IndexInt or PSInt

BCInt is used to store byte counts. CountInt is used to store other counts, such as the number of elements in a dimension of an Igor wave. IndexInt is used to store array indices including wave point numbers. PSInt (pointer-sized int) is used for integers that are used to store a pointer. These types are differentiated only to make the code better express the intent of the programmer.

Your job is to examine each long in your program and change it to the appropriate data type. We did this during the creation of IGOR64 and it resulted in thousands of changes. Fortunately most XOPs are much simpler and will require few changes.

Replacing longs in XOPEntry, RegisterFunction and SetXOPResult Calls

When Igor sends a message to your XOP you often return a result using the **SetXOPResult** XOPSupport routine. In the case of the FUNCTION message, where Igor is asking for the address of your external function, the result is a pointer, specifically a pointer to your function. In many other cases the result is an int.

In IGOR32, pointers and ints are the same size but in IGOR64 pointers are 64 bits and ints are 32 bits. Consequently the data type returned to Igor via SetXOPResult had to be changed so that it is capable of holding a pointer. XOP Toolkit 6 introduces a type for this purpose: **XOPIORecResult**. **XOPIORecResult** is a typedef for SInt32 in IGOR32 and SInt64 in IGOR64.

You must change your XOPEntry routine to return **XOPIORecResult**. For example, in the SimpleFit sample XOP we changed this:

```
extern "C" void
XOPEntry(void)
{
    long result = 0;

    switch (GetXOPMessage()) {
        case FUNCADDRS:
            result = RegisterFunction();
            break;
    }
    SetXOPResult(result);
}
```

to this:

```
extern "C" void
XOPEntry(void)
{
    XOPIORecResult result = 0;

    switch (GetXOPMessage()) {
        case FUNCADDRS:
            result = RegisterFunction();
            break;
    }
    SetXOPResult(result);
}
```

You must also change your RegisterFunction routine, if you have one, to return **XOPIORecResult**. For example, in the SimpleFit sample XOP we changed this:

```
static long
RegisterFunction()
{
    int funcIndex;

    funcIndex = GetXOPItem(0);
    switch (funcIndex) {
        case 0:
            return (long)SimpleFit;
            break;
    }
    return NULL;
}
```

to this:

```
static XOPIORecResult
RegisterFunction()
{
    int funcIndex;

    funcIndex = (int)GetXOPItem(0);
    switch (funcIndex) {
        case 0:
            return (XOPIORecResult)SimpleFit;
            break;
    }
    return 0;
}
```

GetXOPMessage and GetXOPItem Calls

Igor sends your XOP a message by calling your XOPEntry routine which calls **GetXOPMessage** to get the message. In XOP Toolkit 6, the result type of **GetXOPMessage** is changed from long to int. So you need to change your calls to **GetXOPMessage** routine from:

```
long message = GetXOPMessage();
```

to:

```
int message = GetXOPMessage();
```

When Igor sends you a message, you sometimes need to call **GetXOPItem** to get additional information. Sometimes the additional information is in the form of a pointer. Thus, the result type of **GetXOPItem** must be big enough to hold a pointer.

In XOP Toolkit 6, the result type of **GetXOPItem** is changed from long to **XOPIORecParam**. **XOPIORecParam** is defined as **SInt32** in **IGOR32** and **SInt64** in **IGOR64**.

This does not cause a problem except that, when compiling for 64 bits, the compiler will issue a warning on a statement like this:

```
int funcIndex = GetXOPItem(0);
```

The warning will tell you that the 64-bit value returned by **GetXOPItem** is being stored in a 32-bit variable (**funcIndex**) which can possibly cause data truncation. In this case, you know that truncation is not an issue because a **funcIndex** is a small integer. To prevent the warning, you must change this code to:

```
int funcIndex = (int)GetXOPItem(0);
```

This tells the compiler that you know what you are doing when you assign a 64-bit value to a 32-bit variable.

If you already have a cast to another type, leave it as is. For example, this code from WindowXOP1 must not be changed:

```
XOP_WINDOW_REF windowRef;
windowRef = (XOP_WINDOW_REF)GetXOPItem(0);
```

printf and sprintf Issues

Functions in the printf family are problematic. They take a variable number of parameters and determine the type of the parameters using the first parameter which is a format string. If the type specified in the format string does not match the type of the actual parameter passed to the function, a crash can result.

When porting to 64 bits, you must examine every use of printf and sprintf and make sure that the format string and the actual parameters match. For example, suppose your old code looks like this:

```
long numPoints = WavePoints(waveH);
char buf[64];
sprintf(buf, "Number of points: %ld", numPoints);
```

Now, for 64-bit compatibility, you change it to:

```
IndexInt numPoints = WavePoints(waveH);
char buf[64];
sprintf(buf, "Number of points: %ld", numPoints); // BUG
```

This creates a bug. The "%ld" format is correct when compiling as 32 bits because IndexInt is a typedef for long. But it is incorrect when compiling for 64 bits because IndexInt is a typedef for long long (a 64-bit signed integer). This kind of bug will produce incorrect results and can very easily cause a crash.

Here is one way to fix this so that it works on both 32 bits and 64 bits:

```
IndexInt numPoints = WavePoints(waveH);
char buf[64];
sprintf(buf, "Number of points: %lld", (SInt64)numPoints);
```

This technique insures that the value passed to sprintf is 64 bits whether you are compiling as 32 bits or 64 bits. It passes the correct format string, "%lld", to sprintf for a 64-bit integer value.

Be sure to inspect every call to printf and sprintf and make sure that they are correct for both 32 bits and 64 bit compilations.

Command Help for 64-bit XOPs

You provide command help (help for external operations and functions) by providing a suitably-named Igor help file. Igor looks for a help file in the same directory as the XOP whose name is specified by your STR#,1101 resource. If you don't have a STR#,1101 resource, Igor generates a default help file name based on the XOP name. For example, if your XOP is named MyXOP.xop, Igor would look for a file named "MyXOP Help.ihf". This is described under **Help for External Operations and Functions** on page 220.

If you have a 32-bit and a 64-bit version of an XOP, you need to provide only one help file. To support using just one help file, Igor makes several attempts to find the help file associated with the XOP.

Assume you have the following arrangement on your hard disk:

```
MyXOP.xop
MyXOP64.xop
MyXOP Help.ihf
```

In this case, you would put a shortcut for MyXOP.xop in the "Igor Extensions" folder and a shortcut for MyXOP64.xop in the "Igor Extensions (64-bit)" folder.

If the XOP contains an STR#,1101 resource specifying the help file name, "MyXOP Help.ihf", Igor will use this name and find the help file.

Chapter 11 — 64-bit XOPs

If there is no STR#,1101 resource or Igor does not find the help file specified by the resource, when looking for command help for MyXOP64.xop, Igor will first look for a help file named "MyXOP64 Help.ihf" but will not find it. It will then look for "MyXOP Help.ihf", will find it, and will extract the command help from that file.

Now assume you have put your XOPs in the "Igor Pro User Files" folder, arranged like this:

```
Igor Pro User Files
  Igor Extensions
    MyXOP.xop
    MyXOP Help.ihf
  Igor Extensions (64-bits)
    MyXOP64.xop
```

If the XOP contains an STR#,1101 resource specifying the help file name, "MyXOP Help.ihf", Igor will use this name and find the help file. When running MyXOP64.xop, Igor will first look for the help file in "Igor Extensions (64-bits)" and will then look for it in "Igor Extensions".

If there is no STR#,1101 resource or Igor does not find the help file specified by the resource, when looking for command help for MyXOP64.xop, Igor will first look for a help file named "MyXOP64 Help.ihf" in "Igor Extensions (64-bits)" and will not find it. It will then look for "MyXOP Help.ihf" in "Igor Extensions (64-bits)" and again will not find the file. It will then look in "Igor Extensions" instead of "Igor Extensions (64-bits)", first looking for "MyXOP64 Help.ihf" and then looking for "MyXOP Help.ihf". This last attempt will succeed and Igor will extract the command help from that file.

Stated generally, if Igor fails to find the help file in the folder containing the XOP, and if the XOP name exclusive of the extension ends with "64" or "-64", Igor tries again after removing the "64" or "-64". If this fails, and if the name of the folder containing the XOP ends with " (64-bit)", Igor tries again after removing the " (64-bit)" from the folder name.

The reverse is not true. When looking for command help for a 32-bit XOP (e.g., MyXOP.xop), Igor does not search the 64-bit folder.

Other Programming Topics

Macintosh Memory Management

Igor uses Macintosh-style memory management, even when running on Windows. Because XOPs need to access Igor data, such as string parameters and wave notes, 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 or IGOR64.lib, with which all Windows XOPs link.

In Igor Pro 6 and before wave handles are regular Macintosh-style handles. In Igor Pro 7 and later wave handles use different memory management techniques. Consequently the routines described in this section must not be used on wave handles. Instead use wave-specific XOPSupport routines.

In dealing with your own private data, you can use Macintosh routines, standard C/C++ memory management routines, or Windows memory management routines. Using the standard C/C++ 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 Igor string values 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 435.

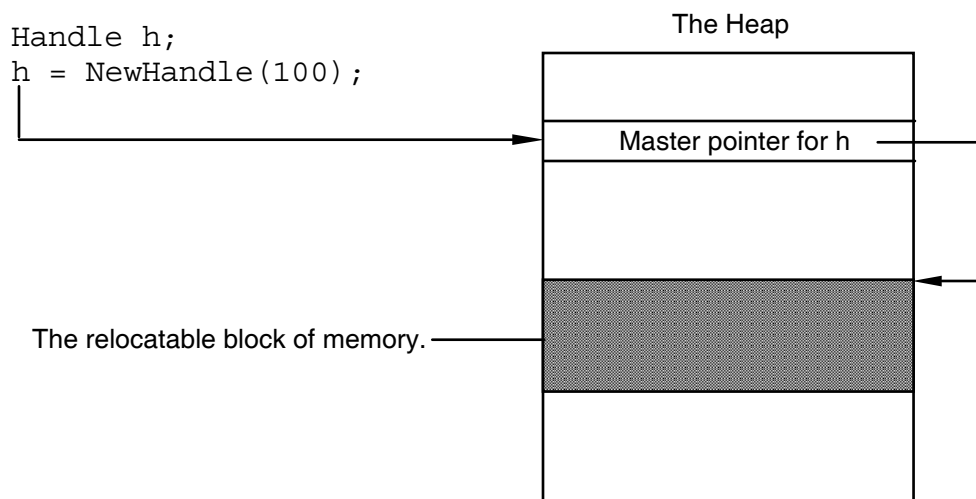
Routine	What It Does
void* NewPtr (BCInt size);	Allocates a block of memory in the heap and returns a pointer to it.
BCInt GetPtrSize (Ptr p);	Returns the number of bytes in the block of memory pointed to by the pointer.
void SetPtrSize (Ptr p, BCInt size);	Sets the number of bytes in the block of memory pointed to by the pointer.
void DisposePtr (void* p);	Frees the block of memory pointed to by the pointer.
Handle NewHandle (BCInt size);	Allocates a block of memory in the heap and returns a handle to it.
BCInt GetHandleSize (Handle h);	Returns the number of bytes in the block of memory referred to by the handle.
void SetHandleSize (Handle h, BCInt size);	Sets the number of bytes in the block of memory referred to by the handle.

Routine	What It Does
void DisposeHandle (Handle h);	Frees the block of memory pointed to by the handle.
int MemError (void);	Returns an error code from the preceding Macintosh memory management call.

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. Using Macintosh memory management, 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.

Handles were invented for the original Macintosh OS to provide a way for the operating system to defragment memory in a heap shared by all applications. The OS would move relocatable blocks if needed to satisfy a new memory allocation request. There were two situations in which the block of memory for a given handle could be moved:

- To compact memory so that another block of memory could be allocated or resized
- To find room to increase the size of the given handle's block when it is resized

Modern Mac OS X programs no longer use handles. But Igor and the XOP Toolkit still use handles and this can not be changed without breaking all existing XOPs. So, for the foreseeable future, Igor and the XOP Toolkit will continue to use handles.

On Mac OS X the block of memory referenced by a handle is no longer relocated to compact memory but it still can be relocated when that handle's block is resized to a larger size.

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 illustrate the difference between pointers and handles. Both snippets allocate a block of

2 bytes of memory, set the first byte to the value 0 and the second byte to the value 1, and then dispose the block.

```
Ptr p;
p = (Ptr)NewPtr(2);
if (p == NULL)
    return NOMEM;
*p = 0;
*(p+1) = 1;
DisposePtr((Ptr)p);

Handle h;
Ptr p;
h = NewHandle(2);
if (h == NULL)
    return NOMEM;
p = *h;           // DEREFERENCE
*p = 0;
*(p+1) = 1;
DisposeHandle(h);
```

Using Handles

In the second snippet above, once `h` is dereferenced, `p` points to the block of memory in the heap. In the original Macintosh OS, the OS could relocate the block of memory to satisfy an allocation request for a different handle or pointer, leaving `p` pointing to the previous location of the block. In this event `p` was said to be a “dangling pointer”.

Avoiding dangling pointers required care on the part of the programmer to guarantee that `p` would remain valid during its use. This often involved “locking” the handle which prevented the operating system from relocating the block. The Macintosh routines `HLock`, `HUnlock`, `HGetState` and `HSetState` were used to lock and unlock handles.

Now that the OS no longer relocates blocks to satisfy allocations of other blocks, the problem of dangling pointers caused by automatic relocation is greatly reduced. There is no longer a need to lock and unlock handles and the `HLock`, `HUnlock`, `HGetState` and `HSetState` routines are not needed. Also no longer needed is the `MoveLockHandle` XOPSupport routine which called `HLock`. If you have old XOP code that uses any of these routines, remove them.

Resizing Handles

The problem of dangling pointers is not completely gone. If you increase the size of a handle by calling **SetHandleSize**, it may not be possible to increase the size of the referenced heap block in place. This occurs if the memory following the heap block is in use. In this situation, `SetHandleSize` relocates the heap block and adjusts the handle’s master pointer accordingly. This means that any time you increase the size of a handle you must re-dereference it.

You also need to re-dereference a handle if you decrease its size. `SetHandleSize` ultimately calls the C `realloc` function or its moral equivalent. `realloc` can relocate a block of memory any time it is called. Although it is relatively rare, some implementations may relocate memory when a block’s size is decreased to reduce memory fragmentation.

NOTE: Any time you increase or decrease the size of a handle, you must re-dereference it.

For example:

```
Handle h;
Ptr p;
int err = 0;

h = NewHandle(2);
```

```
if (h == NULL)
    return NOMEM;
p = *h;                      // DEREFERENCE
*p = 0;
*(p+1) = 1;

SetHandleSize(h,4);          // Increase size of handle
if (MemError() == 0) {       // Make sure resize succeeded
    p = *h;                  // RE-DEREFERENCE
    *(p+2) = 2;
    *(p+3) = 3;
}
else {
    err = NOMEM;             // SetHandleSize failed
}

DisposeHandle(h);
return err;
```

The call to `SetHandleSize` enlarges the block of memory that `h` references. The memory management may be required to move the block in order to enlarge it. If that happens, `p` is left as a dangling pointer - pointing to something that is no longer there. The second dereference of `h` insures that `p` is valid. Without it heap corruption leading to an eventual crash would likely occur.

More subtle is the case where you *indirectly* increase the size of a handle. For example, you might use **ChangeWave**, **MDChangeWave** or **MDChangeWave2** to add elements to a wave:

```
waveHndl waveH = FetchWave("TestWave");
if (waveH != NULL) {
    CountInt numPoints = WavePoints(waveH);
    int waveType = WaveType(waveH);
    void* waveData = WaveData(waveH);    // Dereference waveH

    <Do something with waveData>

    // This call may move the block which would make waveData invalid
    ChangeWave(waveH, 2*numPoints, waveType);

    waveData = WaveData(waveH);          // Dereference waveH again
    <Do something with waveData>
}
```

The call to `ChangeWave` enlarges the block of memory that `waveH` references. The memory management system may be required to move the block in order to enlarge it. If that happens, `waveData` is left as a dangling pointer - pointing to something that is no longer there. The second dereference of `waveH` insures that `waveData` is valid. Without it heap corruption leading to an eventual crash would likely occur.

Obsolete Handle Locking Routines

The following routines for locking and unlocking handles are obsolete and should not be used. You may see them in the source code for old XOPs. If you are revising an old XOP, remove calls to these routines.

Routine	Description
HLock	Locks a handle
HUnlock	Unlocks a handle
HGetState	Gets lock state of a handle
HSetState	Sets lock state of a handle
MoveHHi	Moves a handle to the top of the heap

Routine	Description
MoveLockHandle	Moves it to the top of the heap and locks it
SetWavesStates	Sets lock state for an array of wave handles

Although it is no longer necessary to lock handles, the heap block they reference can still move if you increase the size of the handle using **SetHandleSize**. Thus you must re-dereference a handle after increasing its size as explained under **Resizing Handles** on page 199.

Accessing Igor Data Objects

You might use NewPtr and DisposePtr or NewHandle and DisposeHandle when dealing with your own private data. But 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 platform. The WindowXOP1 sample XOP illustrates both of these techniques. WindowXOP1.cpp has a few platform-dependent ifdefs. The bulk of the platform-dependent code is in WindowXOP1Mac.cpp and WindowXOP1Win.cpp. The public routines provided by these files have the same interface regardless of platform. This relieves the main file, WindowXOP1.cpp, 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:

FullPathPointsToFile	FullPathPointsToFolder	
XOPCreateFile	XOPDeleteFile	
XOPOpenFile	XOPCloseFile	
XOPReadFile	XOPReadFile2	XOPReadLine
XOPWriteFile		
XOPGetFilePosition	XOPGetFilePosition2	
XOPSetFilePosition	XOPSetFilePosition2	

Chapter 12 — Other Programming Topics

XOPNumberOfBytesInFile **XOPNumberOfBytesInFile2**
XOPAtEndOfFile

These routines are defined in XOPFiles.c, XOPFilesMac.c and XOPFilesWin.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 be HFS paths using colons as path separators and on Windows they must use backslashes. Do not use POSIX paths (with forward slashes).

Prior to Igor Pro 6.10, Igor did not support file names longer than 31 characters on Macintosh. To work with long file names on Macintosh, an XOP must be compiled with XOP Toolkit 5.09 or later.

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 MAX_ symbols shown above are defined in IgorXOP.h.

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.

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	Path Type
hd:Folder1:Folder2:File	C:\Folder1\Folder2\File	Full path
:Folder2:File	.\Folder2\File	Partial path

On Mac OS X, POSIX paths with forward slashes are often used. However, Igor and the XOP Toolkit are based on Apple’s Carbon API which does not use POSIX paths. Carbon uses Macintosh HFS (hierarchical file system) paths with colon separators. When we speak of “Macintosh paths”, we are talking about HFS colon-separated paths.

Igor commands can refer to files using paths. HFS paths are recommended but Windows paths are also accepted. 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 HFS 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 HFS syntax when running on Macintosh and Windows syntax when running on Windows. For example, SimpleLoadWaveOperation.cpp 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 HFS 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 **GetNativePath**.

Other cross-platform path-related XOPSupport routines include **ConcatenatePaths**, **GetDirectoryAndFileNameFromFullPath**, **FullPathPointsToFile**, and **FullPathPointsToFolder**.

On Mac OS X you may need to convert an HFS path that you receive as a parameter to a POSIX path in order to call an operating system routine or a standard file library I/O routine. You can do this using the **HFS-ToPosixPath** XOPSupport routine.

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.

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.

NOTE: Adding a dialog to a Macintosh XOP is completely different in XOP Toolkit 7 and later. Consequently, we recommend that you not write new code for adding a dialog on Macintosh using the techniques described in this chapter.

The methodology used by the XOP Toolkit to create a dialog on Macintosh is antiquated. It is based on DLOG and DITL resources and uses Carbon Dialog Manager routines. Though antiquated, it still works and provides a relatively easy way to create a cross-platform Igor-style dialog. In the past you could design a Dialog Manager dialog graphically using ResEdit. ResEdit is no longer supported so you must design these dialogs by entering text in a .r file.

For an example of a cross-platform Igor-style dialog, see the **GBLoadWave** sample XOP.

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++ development system stores the resource ID numbers in the resource.h file associated with each project.

Macintosh Dialogs

Implementing a modal dialog in a Macintosh XOP using the Carbon Dialog Manager API is the same as in a standalone Carbon program with a few exceptions.

An XOP must use the XOPSupport routines **GetXOPDialog**, **DoXOPDialog**, and **DisposeXOPDialog** instead of the corresponding Macintosh routines **GetNewDialog**, **ModalDialog**, and **DisposeDialog**.

XOPs must use Apple Appearance Manager-compatible techniques to create dialogs. This includes creating a dlgs resource with the kDialogFlagsUseControlHierarchy bit set. See the sample XOPs for examples.

Creating the Macintosh resources for an XOP dialog presents a bit of a problem. Apple no longer provides tools for creating dialog resources so you must edit the .r file in a text editor. For more on Macintosh resource creation, **Creating Resources on Macintosh** on page 52.

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++ graphical resource editor. The graphical resource editor is not included in the express editions of Visual C++. For more on Windows resource creation, see **Creating Resources on Windows** on page 52.

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**, **CreatePopupMenu**) 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 using the Carbon Dialog Manager, 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 using the Carbon Dialog Manager, 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.cpp 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.cpp. This highest level routine is implemented 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

Chapter 12 — Other Programming Topics

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.cpp 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 are implemented using the Macintosh menus. On Windows, popup menus are implemented using combo boxes.

The GBLoadWaveDialog.cpp 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.

Chapter 12 — Other Programming Topics

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 **CreatePopupMenu** XOP-Support routine and is disposed when you call **KillPopMenus**.

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.

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 six 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
```

As the user types and clicks in the dialog, the XOP displays the command being generated in the command box, using the **DisplayDialogCmd** XOP-Support routine

When the user clicks Do It, To Cmd, or To Clip, the XOP calls the **FinishDialogCmd** XOP-Support routine, which does the appropriate thing with the command generated by the dialog.

When the user clicks the Help button, the dialog calls **XOPDisplayHelpTopic**. XOPDisplayHelpTopic displays the specified topic in the Igor help file associated with the XOP. For details on creating an Igor help file, see Chapter 13.

The XOPSupport routines include extensive support for popup menus in dialogs. The GBLoadWaveDialog.cpp file illustrates 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. 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, © 2010 WaveMetrics, Inc., all rights reserved."
};

resource 'vers' (2) {                // Version of Igor
    0x06, 0x00, release, 0x00, 0,    // Version bytes and country code
    "6.00",
    "(for Igor Pro 6.00 or later)"
};
```

Windows Version Resources

You create a version resource using the Visual C++ graphical resource editor or using a text 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, eight or sixteen-byte boundaries and how 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. In XOP Toolkit 6 we use pragma statements.

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

Chapter 12 — Other Programming Topics

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:

```
#pragma pack(2)           // All Igor structures are two-byte aligned

<Structures defined here>

#pragma pack()           // Reset structure alignment
```

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 pragma 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.

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

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 88 and **External Function Structure Parameter Example** on page 109.

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 211.

Igor Field Type	C Field Type	Notes
WAVE	waveHndl	Always check for NULL.
NVAR	NVARRec	Use with GetNVAR and SetNVAR.
SVAR	SVARRec	Use with GetSVAR and SetSVAR.
DFREF	DataFolderHandle	Always check for NULL.
FUNCREF	void*	Use with GetFunctionInfoFromFuncRef.
STRUCT	struct	Embedded substructure.
char	char	
uchar	unsigned char	
int16	short	
uint16	unsigned short	
int32	SInt32	
uint32	UInt32	
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.

Likewise, always check for NULL before using a DataFolderHandle field.

Do not access NVARRec and SVARRec fields directly but use the **GetNVAR**, **SetNVAR**, **GetSVAR** and **SetSVAR** XOPSupport routines instead as explained under **NVARs and SVARs In Structures** on page 212. If the calling Igor procedure attempts to access a non-existent global numeric variable or global string variable, the those 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 213 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 parameters 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. A NULL string handle indicates an error in the calling user-defined function and should be treated as an error.

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. In the later case, you must remove the null terminator when you are finished using it as a C string. See **Understand the Difference Between a String in a Handle and a C String** on page 237 for details.

Wave Handles In Structures

Usually a waveHndl field in a structure is used to pass a wave reference from Igor to your XOP. However, it can also be used to pass a wave reference from your XOP to Igor.

A waveHndl field will be NULL if it is uninitialized or if its initialization failed. Always test a waveHndl field before using it. If it is NULL you can return the NULL_WAVE_OP Igor error code.

In Igor Pro 6.20 or later, if you store a wave handle in a waveHndl field of a structure you need to inform Igor by calling **HoldWave** on the structure field. By doing this, you maintain the integrity of Igor's internal wave reference count. For details see **Wave Reference Counting** on page 128.

Data Folder Handles In Structures

Usually a DataFolderHandle field in a structure is used to pass a data folder reference from Igor to your XOP. However, it can also be used to pass a data folder reference from your XOP to Igor.

A DataFolderHandle field will be NULL if it is uninitialized or if its initialization failed. Always test a DataFolderHandle field before using it. If it is NULL you can return the NULL_DATAFOLDER_OP Igor error code.

In Igor Pro 6.20 or later, if you store a data folder handle in a DataFolderHandle field of a structure you need to inform Igor by calling **HoldDataFolder** on the structure field. By doing this, you maintain the integrity of Igor's internal data folder reference count. For details see **Data Folder Reference Counting** on page 140.

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.

```
#pragma pack(2)           // All Igor structures are two-byte aligned

#define kF2StructureVersion 1000      // 1000 means 1.000.
struct F2Struct {                  // Format of structure parameter.
    UInt32 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;

#pragma pack()              // Reset structure alignment

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 != kF2StructureVersion) {
    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

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.

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 15.

Chapter 12 — Other Programming Topics

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.

CallFunction and Igor Threads

You can call a thread-safe user-defined function or external function while you are running in an Igor preemptive thread. You can tell if you are running in a thread by calling **RunningInMainThread**.

When you call GetFunctionInfo, you receive information about the function in a FunctionInfo structure. The FunctionInfo structure contains an isThreadSafe field. If you are running in a thread, you can check the isThreadSafe field to make sure the function you are calling is threadsafe.

If you try to call a non-threadsafe function from a thread using CallFunction, Igor will return an error, typically the ONLY_THREADSafe ("Function must be ThreadSafe") error.

CallFunction and Igor Independent Modules

Independent modules allow advanced programmers to create separately-compiled Igor procedure code modules. See the Igor Pro manual for details. If you are not the author of an Igor independent module you can ignore this section.

When you call **GetFunctionInfo** with a simple function name, such as "MyFunction", the information returned describes the named function in the currently-executing independent module. If you call GetFunctionInfo with a fully-qualified name, such as "MyModule#MyFunction", the information returned describes the named function in the named module. So if you want to call from one independent module to another using **CallFunction**, you must use fully-qualified names when calling GetFunctionInfo.

GetFunctionInfoFromFuncRef does not support cross-independent-module calls. It assumes that the FUNCREF passed to it was created in the module that is currently running. There is currently no way to use a FUNCREF to make cross-module calls. Use **GetFunctionInfo** with a qualified name instead.

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 know 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.
```

```
#pragma pack(2)                // All Igor structures are two-byte aligned
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;
#pragma pack()                 // Reset structure alignment

int
DoOurOperation(waveHndl coeffsWaveH)
{
    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]=WAVE_TYPE; // First parameter is wave
    requiredParameterTypes[1]=NT_FP64;   // Second parameter is a numeric
    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 = coeffsWaveH;
    for(i=0; i<5; i+=1) {
        parameters.x = i;
        if (err = CallFunction(&fi, (void*)&parameters, &result))
            return err;
        values[i] = result;
    }

    return 0;
}
```

The XOPMain Function

When Igor launches an XOP it calls the XOP's main function which is named XOPMain.

Originally the main function of an XOP looked like this:

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

This form was chosen because, back in the dark ages, some compilers required a main function and Igor needed to pass the ioRecHandle to the XOP to facilitate subsequent communications.

This use of main is considered to be an error by recent compilers.

To cope with this change, as of Igor Pro 6.20, Igor looks for either of the following main functions:

```
// Old main - Does not work with recent compilers
HOST_IMPORT int
main(IORecHandle ioRecHandle)
```

```
// New XOPMain - Requires Igor Pro 6.20 or later
HOST_IMPORT int
XOPMain(IORecHandle ioRecHandle)
```

Igor Pro 6.20 and later versions first look for the new XOPMain function and, if that is not present in the XOP, they look for the old main function.

main and XOPMain behave identically.

All new XOPs should use XOPMain, not main.

If you are updating any existing XOP, change it to use XOPMain, not main.

XOPs that use XOPMain require Igor Pro 6.20 or later.

Checking For Aborts

If your external operation or external function might take a long time to finish you will probably want to allow the user to abort it. There are two XOPSupport routines to help with this, **SpinProcess** and **CheckAbortCheckAbort**.

SpinProcess is so-named because originally it spun the beachball cursor during a lengthy calculation on Macintosh. Now the operating system spins the beachball on Macintosh. On Windows, SpinProcess still spins the beachball in the bottom/left corner of the status area.

SpinProcess returns non-zero if an abort has occurred because:

- The user pressed cmd-dot (Macintosh) or Ctrl-Break (Windows)
- The user clicked the Abort button in the status area (Windows)
- A procedure called the Abort operation

Otherwise SpinProcess returns zero.

You can call SpinProcess from a lengthy loop like this:

```
int err = 0;
for(i=0; i<1E9; i+=1) {                // This will take a long time
    <do one iteration of your calculation>
    if ((i % 100) == 0) {                // Do this every 100 iterations
        if (SpinProcess() != 0) {
            err = -1;                    // -1 signals an cancel or abort
            break;
        }
    }
}
return err;
```

SpinProcess does a callback to Igor which can take some time. That's why we call SpinProcess every 100 iterations instead of every iteration. You can adjust the number 100 to suit your situation. To achieve reasonable responsiveness, SpinProcess should be called at least a few times per second if practical.

SpinProcess is thread-safe only if you are running with Igor Pro 6.20 or later. If your external operation or function is thread-safe and you are running in a thread and you are running with an older version of Igor, you can use CheckAbort instead of SpinProcess, as shown below.

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 64 for details.

CheckAbort checks for the user pressing the abort key combinations at the moment you call it and does not check for the other abort conditions listed above. CheckAbort does not do a callback to Igor and is therefore faster than SpinProcess and does not introduce the possibility of recursion. Also CheckAbort is thread-safe with any version of Igor.

On Macintosh CheckAbort may not work correctly on non-English keyboards. It checks the key that is the dot key on an English keyboard. This is the key that is two keys to the left of the righthand shift key. On a French keyboard, for example, that is the colon key so you must press cmd-colon instead of cmd-dot.

If this is a problem, you may prefer to use the CheckEventQueueForUserCancel function from the Carbon library instead of CheckAbort. However CheckEventQueueForUserCancel checks for the escape key in addition to cmd-dot. It also removes events from the event queue which may prevent Igor from detecting that you want to abort procedure execution.

Here is the previous example reworked to call CheckAbort instead of SpinProcess if necessary:

```
int err = 0;
int callSpinProcess = igorVersion<6.20 && !RunningInMainThread();
for(i=0; i<1E9; i+=1) { // This will take a long time
    <do one iteration of your calculation>
    if ((i % 100) == 0) { // Do this every 100 iterations
        if (callSpinProcess) {
            if (SpinProcess() != 0) {
                err = -1; // -1 signals an cancel or abort
                break;
            }
        }
        else {
            if (CheckAbort(0) != 0) {
                err = -1; // -1 signals an cancel or abort
                break;
            }
        }
    }
}
return err;
```

Writing XOPs in C++

Writing an XOP in C++ is not significantly different from writing it in C.

Calling C Code from 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 routines. 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 "C" declarations when you use XOPSupport routines in your C++ files.

C++ Code Called From C Code

Since Igor must work with both C and C++ XOPs, it uses C calling conventions when calling an XOP function. Thus your XOPMain function, XOPEntry function, and any external operation or external function execute routines should be declared extern "C". For example:

```
HOST_IMPORT int          // HOST_IMPORT includes extern "C" in C++ code
XOPMain(IOPRecHandle ioRecHandle)

extern "C" void          // XOPEntry routine
XOPEntry(void)

extern "C" int           // External operation execute routine
ExecuteXOP1(XOP1RuntimeParamsPtr p)

extern "C" int           // External function execute routine
XFUNC1Div(XFUNC1DivParams* p)
```

In C code you would have static instead of extern "C". Although it is not technically correct, using static in a C++ XOP also works with all of the C++ compilers that we have seen.

Using C++ Exceptions

In implementing C++ exceptions, it is very important that your final catch block always be before code execution returns to Igor. All exceptions thrown from within your XOP must be caught inside your XOP. This includes exceptions thrown by C++ library routines that you call.

Providing Help

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
- Igor tips
- Windows status line 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 must have a ".ihf" extension. After editing the file as an Igor formatted notebook, the next time you open it as a help file, Igor will "compile" it.

The help compiler generates a list of topics and subtopics and their locations in the help file. Its output is stored in the help file itself. You can compile your help file on either platform and it will work on both.

Your help file should include an overview, installation instructions, 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. 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.

If you create a 32-bit and a 64-bit version of your XOP, you can create one help file for both versions. For details, see **Command Help for 64-bit XOPs** on page 195.

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.

By default Igor 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 looks 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.

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

Prior to Mac OS X, Igor used the Macintosh Balloon Help system and XOPs could add balloon help for menu and dialog items. On Mac OS X, this no longer works. If you have hmnu and hdlg resources in old

XOP source code you can remove them. The Macintosh hmnu resource is distinct from the Windows HMNU resource described under **Status Line Help For Windows XOP Menu Items** on page 221.

It is still possible to supply an Igor tip for individual menu items as described in the next section.

Igor Tips 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 Igor tip 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 as Igor uses the resource name to find a help resource for a given menu item. However, 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 sub-menus, 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

Chapter 13 — Providing Help

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.

General Help For XOP Dialogs and Windows

The user also may need help that explains the general 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 14

Debugging

Overview

In this chapter we present tips and techniques learned through years of XOP programming that may save you valuable time.

Most bugs can be avoided by careful programming. You can dramatically 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 proof-read 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 intermittent bugs that are very difficult to fix. 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.

```
int numBytes;
double* dp;

if (<condition1>)
    numBytes = 100;
else
    if (<condition2>)
        numBytes = 200;

dp = (double*)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 a PowerPC 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;
    CountInt dims[MAX_DIMENSION_SIZES+1];

    MemClear(dims, sizeof(dims));
    dims[ROWS] = 100;
    dims[COLUMNS] = 100;
    MDMakeWave(&waveH, "wave0", NULL, dims, NT_FP32, 1);
}
```

```
    waveData = (char*)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;
    CountInt 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 = (char*)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(int* ip)
{
    *ip = 0;
}

void
F2_BAD(void)
{
    int* ip1;

    F1(ip1);                // Bug
}
```

The variable ip1 is a pointer to an int so the compiler is happy with this code. At runtime, however, it may cause a crash. The problem is that ip1 is an uninitialized variable. It contains a random value that could point to any location in memory. When F2_BAD calls F1, F1 sets the value pointed to by ip1 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 ip1.

This example shows two correct ways to call F1.

```
void
F2_GOOD(void)
{
    int int1;
    int ip1;
```

```

    F1(&int1);
    ip1 = (int*)NewPtr(sizeof(int));
    if (ip1 != NULL) {
        F1(ip1);
        DisposePtr((Ptr)ip1);
    }
}

```

In the first call to F1, we pass the address of the local variable int1 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 (&int1 in F2_GOOD) or to some space in the heap (ip1 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 with dire consequences. Here is an example.

```

int
Test(int v1, int v2)
{
    Handle h;
    BCInt 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;
    BCInt 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)
{
    BCInt size;

    size = GetHandleSize(gH);    // Uses global before it is initialized.
    memset(**gH, 0, size);
}

void
InitializePart2(void)
{
    gH = NewHandle(100);        // Initializes global handle.
}

void
XOPMain(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. It is good practice to 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 or Xcode's Leaks performance tool. 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.

If you find a memory leak you will also discover that it can be very difficult to track down. It is much better to code carefully in the first place than to spend your time chasing leaks. Coding carefully in this case means having a clear idea of when memory is allocated and when it is deallocated, avoiding complicated schemes and *carefully proofreading* your code after you write it.

Disposing Memory Blocks That Don't Belong to You

Wave and data folder handles belong to Igor, not you. Thus you should never dispose them. 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). Check the documentation and understand who owns each memory resource.

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 camouflage bad programming practices. Good code still always checks memory allocations and handles failed allocations gracefully.

Dangling Pointers

A dangling pointer is a pointer that used to point to a block of memory in the heap but no longer points to a valid block because the block has been freed. For example:

```
int
Bad()
{
    Ptr p;
    p = (Ptr)NewPtr(100);
    if (p == NULL)
        return NOMEM;
    memset(p, 0, 100);
    . . .
    DisposePtr((Ptr)p);    // Free the block pointed to by p
    . . .
    memset(p, 1, 100);    // BUG: p is a dangling pointer
    . . .
    return 0;
}
```

In this example the bug is obvious. But in a lengthy and complex function it is often not obvious. And if you store the pointer in a global and use it from several different functions, it becomes much harder to keep track of when it is valid.

This bug may cause an immediate crash or may cause a crash later or may be asymptomatic. It depends on what happens to be at the memory location referenced by *p* after the block is freed. This can create irreproducible bugs that are nasty to debug. So it is well worth your while to use good coding practices and to *carefully proofread your code* to avoid this kind of problem in the first place.

A good technique is to use the pointer itself as a flag to determine if it is valid or not:

```
int
Good()
{
    Ptr p;
    p = (Ptr)NewPtr(100);
    if (p == NULL)
        return NOMEM;
    memset(p, 0, 100);
    . . .
    DisposePtr((Ptr)p);    // Free the block pointed to by p
    p = NULL;              // Set to NULL to signify no longer valid
    . . .
    if (p == NULL) {       // Test pointer before using
        p = (Ptr)NewPtr(100);
        if (p == NULL)
            return NOMEM;
    }
    memset(p, 1, 100);
    . . .
    DisposePtr((Ptr)p);
    p = NULL;              // Set to NULL to signify no longer valid
}
```

```

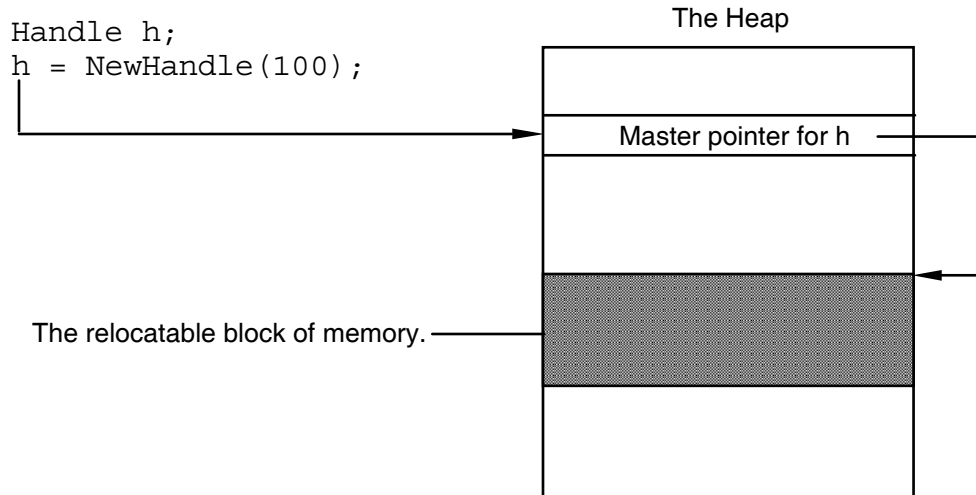
    return 0;
}

```

The last `p=NULL` statement seems unnecessary because `p` is a local variable in this example and will cease to exist when the function returns. In a complex real-world function, it may not be so obvious so it is a good idea to consistently clear the pointer when the block it references is freed. If the pointer is a global variable, consistently clearing it is even more important.

Handles and Dangling Pointers

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.



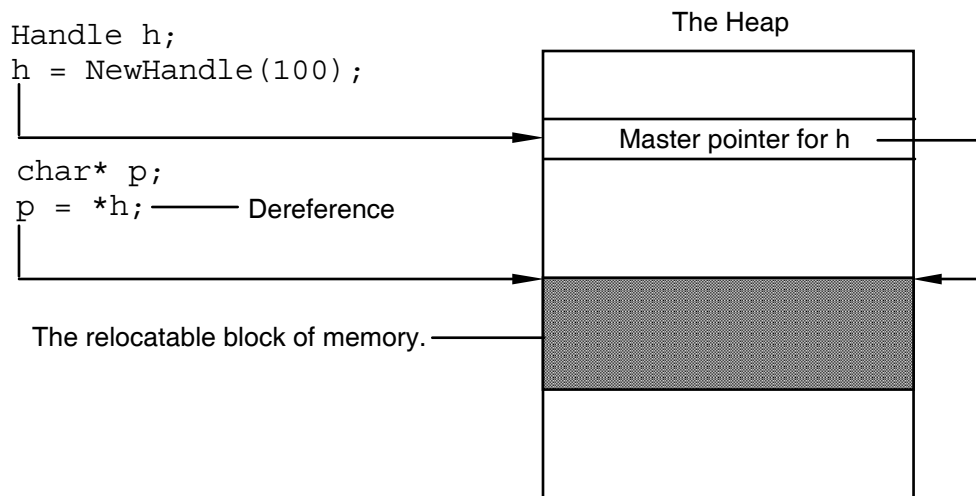
On Mac OS 9, the operating system could relocate the block of memory any time any memory allocation was made by any routine. This allowed the OS to maximize use of limited memory by consolidating small free blocks. On Mac OS X, the operating system will not relocate a block unless relocation is required to satisfy a call to increase the size of that specific block. Thus relocation is far more rare on Mac OS X but can still happen.

Windows XOPs use Macintosh-style handles, as is explained in the section **Data Sharing** on page 65. Igor's emulation of the Macintosh Memory Manager on Windows works like Mac OS X. A block will not be relocated unless it is required to satisfy a call to increase the size of that specific block.

As the following sections explain, you must be careful when using a handle to avoid using a dangling pointer to the handle's heap block.

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 in order to satisfy a request to increase the size of the block, 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 referenced by a different handle or pointer. If we use `p` to read data, we will read garbage. If we use `p` to write data, we may trash a different block, 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.

The memory manager will not move the block of memory capriciously. It will only move the block if you try to increase its size using `SetHandleSize` either directly or indirectly. Thus you can avoid a dangling pointer bug by re-dereferencing the handle after resizing it. For example:

```
Handle h;
char* p;

h1= NewHandle(100);
p = *h;                      // Dereference h to point to data.
memset(p, 0, 100);
. . .
SetHandleSize(h, 200);      // Increase size of heap block
if (MemError())
    return NOMEM;
p = *h;                      // Re-dereference h to point to data.

<Use p to access data>;
```

We need to dereference `h` the second time because the second call to `SetHandleSize` may cause the memory manager to relocate the block of memory that `h` refers to.

Dereferencing Wave Handles

Here is a less obvious example of a dangling pointer, this time involving a wave handle:

```
waveHndl waveH;
double* dp;
waveH = FetchWave("wave0");
if (waveH == NULL)
    return NOWAV;
<Check wave data type and number of points here>
dp = (double*)WaveData(waveH);  // Dereference wave handle
. . .
```



```
XOPCommand(<some command>);      // Call Igor to execute a command
dp[0] = 0;                        // Possible BUG
```

The bug here is that the XOPCommand call to Igor could possibly increase the size of the wave, for example, by executing Redimension on the wave. If that happened the wave's heap data could be relocated. Then dp would be a dangling pointer and the assignment statement using it could cause a crash either immediately or later.

The problem would be even more difficult to recognize if the XOPCommand call were in a subroutine called by the code above.

This bug would be asymptomatic if the wave were not resized by the Igor command or if the system was able to resize the heap block without relocating it because the memory following the heap block happened to be free. Once in a great while, however, relocation would occur, with unpredictable results. In other words, this bug would cause a highly intermittent and irreproducible symptom and would be nasty to track down.

The solution is to redereference the wave handle after any call that conceivably could resize the wave. To do this, add another call to WaveData:

```
XOPCommand(<some command>);      // Call Igor to execute a command
dp = (double*)WaveData(waveH);   // Re-dereference wave handle
dp[0] = 0;
```

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, if possible. 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.

This technique can not be used with wave handles because you must use the WaveData function to dereference them.

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.

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, as in the case of dereferencing a wave handle, re-dereference the pointer after calling *any* subroutine that could conceivably resize the block.

Debugging Techniques

This section discusses several methods for debugging an XOP:

- Using a symbolic debugger
- Using XOPNotice (equivalent to a Print statement)
- Using a crash log
- Other debugging tricks

This section does not cover the most effective debugging method: *carefully proofreading 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 and eventually zero-in on the problem.

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 under 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(waveH));  
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.

If you call XOPNotice from an Igor preemptive thread, Igor buffers the message and prints it in the history at a later time from the main thread. This means that messages from different threads may appear in the history in an unpredictable order.

Other Debugging Tricks

In rare cases, you may want to know how your external operation or function was called.

In Igor Pro 6 or later, a simple way to do this is to invoke Igor's debugger, like this:

```
XOPSilentCommand("Debugger");
```

If debugging is enabled (via the Procedure menu), Igor will break into the debugger after executing the current user-defined function statement – that is, after your external function or operation returns.

If no procedure is running in Igor, this statement does nothing.

You can also invoke Igor's DebuggerOptions operation using XOPSilentCommand.

You can determine the function or chain of functions that called your external function or operation using the **GetIgorCallerInfo** or **GetIgorRTStackInfo** callbacks.

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 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, which are not limited in 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, use it as a C string, and remove the null character. This is useful for cases where you can not set an upper limit on the length of the string. For example:

```
Handle h;
int len;
char* p;

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.  
    p = *h;  
    <Use p as a C string>;  
    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.

Understand Who Owns a Handle

In some cases Igor is the owner of a handle and is responsible for disposing it. In other cases, you own it and are responsible for disposing it. The XOP Toolkit documentation will tell you which is which.

For example, when your external operation receives a string parameter in a handle, Igor owns it and you must not dispose of it. By contrast, when your external function receives a string parameter in a handle, you own it and must dispose of it or pass it back to Igor as a string result in which case Igor receives ownership of it.

In the case of wave and data folder handles, Igor always owns these. You must never dispose of them or otherwise directly modify them except that you can modify the wave data as explained under **Routines for Accessing Waves** on page 119.

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 72 and **Choose a Distinctive Function Name** on page 100 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 148 for details.

Watch Out for Recursion

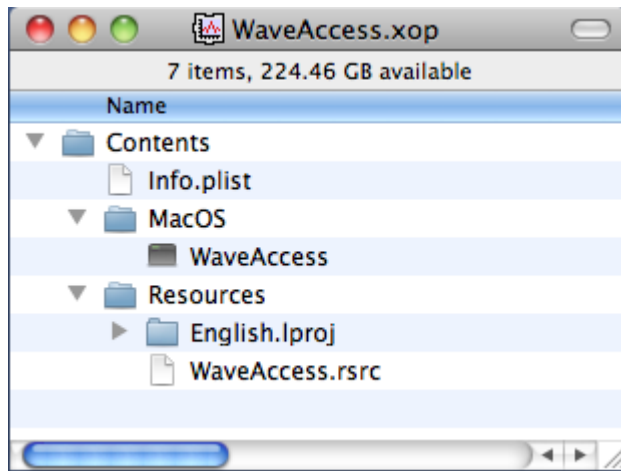
If your XOP adds a window to Igor, the DoUpdate, XOPCommand, XOPSilentCommand, and SpinProcess XOPSupport routines can cause Igor to call your XOP while your XOP is already running. See **Handling Recursion** on page 64 for details.

Structure Alignment

Make sure that all structures that might be shared with Igor are 2-byte-aligned. See **Structure Alignment** on page 209 for details.

Check Your Info.plist File

As explained under **XOP Projects in Xcode** on page 32, Macintosh XOP is a “package”. That is, it is not a simple file but rather a folder with a particular structure. To see the contents of the package folder, right-click it and choose Show Package Contents. You will see something like this:



In order for the XOP to load, the package name (e.g., **WaveAccess.xop**) must agree with the executable name (e.g., Contents/MacOS/**WaveAccess**) which must agree with the value of the CFBundleExecutable key in the Info.plist file (Contents/Info.plist).

The package's Info.plist file is an XML file that provides information about the package to the operating system. You can view it in a text editor or in Apple's Property List Editor program. One of the keys in the XML file is named CFBundleExecutable. In Property List Editor the CFBundleExecutable key may be listed as "Executable file".

If the XOP name is WaveAccess, the package must be named WaveAccess.xop, the executable file in the Contents/MacOS folder of the package folder must be named WaveAccess and the CFBundleExecutable key in the Info.plist file must be set to WaveAccess. If any of these names are incorrect, Igor will be unable to load the XOP and usually will report "Can't load the executable file in the XOP package".

You can set the CFBundleExecutable key either directly in the Info.plist file or using the Executable setting. In Xcode it is in the Info tab of the target settings.

XOPSupport Routines

Table of Contents

About XOPSupport Routines.....	242
Routines for Communicating with Igor.....	243
Operation Handler Routines	246
Routines for Accessing Waves	257
Routines for Accessing Variables.....	289
Routines for Accessing Data Folders.....	298
Routines for XOPs with Menu Items.....	314
Routines for XOPs that Have Dialogs	325
Routines for XOPs that Access Files.....	338
Routines for File-Loader XOPs	353
Routines for XOPs with Windows.....	357
Routines for XOPs with Text Windows	363
Routines for Dealing with Resources	376
Routines for XOPs That Use FIFOs	378
Numeric Conversion Routines.....	380
Routines for Dealing With Object Names	384
Color Table Routines	390
Command Window	393
Routines for Dealing With Igor Procedures	397
Multithreading Support Routines.....	406
Windows-Specific Routines	409
Miscellaneous Routines	412
Programming Utilities.....	427
Macintosh Emulation Routines.....	435

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. This release of the XOP Toolkit supports Igor Pro 6.00 or later so issues involving earlier versions of Igor are not discussed below. Routines that require a later version of Igor are so noted. See **Igor/XOP Compatibility Issues** on page 66 for further information on compatibility.

Routines for Communicating with Igor

These routines provide communication between your XOP and Igor.

XOPInit

```
void
XOPInit(ioRecHandle)
IORecHandle ioRecHandle;    // Used to pass info between Igor and XOP
```

The XOP must call XOPInit from its XOPMain function, 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.

Thread Safety

XOPInit is not thread-safe.

SetXOPType

```
void
SetXOPType(type)
int 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 and later 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 164 for further information.

See **The IORecHandle** on page 62 for details.

Thread Safety

SetXOPType is not thread-safe.

SetXOPEntry

```
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 XOPMain function, passing it the address of your XOPEntry routine.

Thread Safety

SetXOPEntry is not thread-safe.

SetXOPResult

```
void
SetXOPResult(result)
int 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 57 for details on error codes.

Thread Safety

SetXOPResult is not thread-safe.

GetXOPResult

```
XOPIORecResult
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.

Thread Safety

GetXOPResult is not thread-safe.

SetXOPMessage

```
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.

Thread Safety

SetXOPMessage is not thread-safe.

GetXOPMessage

```
int
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 63 for details.

Thread Safety

GetXOPMessage is not thread-safe.

GetXOPStatus

```
int  
GetXOPStatus(void)
```

NOTE: This routine is obsolete and may be removed in a future version of the XOP Toolkit.

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 62.

Thread Safety

GetXOPStatus is not thread-safe.

GetXOPItem

```
XOPIORecParam  
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 63 for details.

Thread Safety

GetXOPItem is not thread-safe.

Operation Handler Routines

XOPs implement external operations using Igor's Operation Handler as described in Chapter 5, **Adding Operations**. This section documents XOPSupport routines used in conjunction with Operation Handler.

RegisterOperation

```
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.

The options parameter is kOperationIsThreadSafe for a thread-safe operation or 0 otherwise.

Returns 0 or an error code.

Example

```
char* cmdTemplate;           // From SimpleLoadWaveOperation.cpp
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), (void*)ExecuteSimpleLoadWave, 0);
```

Thread Safety

RegisterOperation is not thread-safe.

SetOperationNumVar

```
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.

Example

```
SetOperationNumVar("V_VDT", number);    // From VDTOperations.cpp
```

Thread Safety

SetOperationNumVar is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetOperationStrVar

```
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 is a null-terminated C string and points to the value to be stored in the named variable.

Use **SetOperationStrVar2** if your data is binary (may contain bytes whose value is zero other than the null terminator).

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 local variable.

Returns 0 or an error code.

Thread Safety

SetOperationStrVar is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetOperationStrVar2

```
int
SetOperationStrVar2(varName, data, dataLength)
const char* varName;    // C string containing name of variable
const char* data;       // Value to store in variable
int dataLength;         // Length of data in bytes
```

SetOperationStrVar2 is just like **SetOperationStrVar** except for the dataLength parameter. Use SetOperationStrVar2 if the data is binary and may contain null bytes (bytes whose value is zero).

Added for Igor Pro 6.10B02. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

Thread Safety

SetOperationStrVar2 is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

VarNameToDataType

```
int
VarNameToDataType(varName, dataTypePtr)
const char* varName;    // Pointer to varName field in runtime param struct
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 get or set the value of that variable. An example is Igor's Open operation which takes a "refNum" parameter and stores a value in the referenced variable.

After calling VarNameToDataType you would then call **FetchNumericDataUsingVarName**, **StoreNumericDataUsingVarName**, **FetchStringDataUsingVarName** 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.

See **VarName Parameters** on page 83 for further information.

Example

See VDTReadBinary.cpp.

Thread Safety

VarNameToDataType is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

FetchNumericDataUsingVarName

```
int
FetchNumericDataUsingVarName(varName, realPartPtr, imagPartPtr)
const char* varName;    // Pointer to varName field in runtime param struct
double* realPartPtr;
double* imagPartPtr;
```

Retrieves data from 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 get the value of that variable.

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.

See **VarName Parameters** on page 83 for further information.

Added for Igor Pro 6.10. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

Thread Safety

FetchNumericDataUsingVarName is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

StoreNumericDataUsingVarName

```
int
StoreNumericDataUsingVarName(varName, realPart, imagPart)
const char* varName;    // Pointer to varName field in runtime param struct
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.

See **VarName Parameters** on page 83 for further information.

Example

See VDTReadBinary.cpp.

Thread Safety

StoreNumericDataUsingVarName is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

FetchStringDataUsingVarName

```
int
FetchStringDataUsingVarName(varName, hPtr)
const char* varName;    // Pointer to varName field in runtime param struct
Handle* hPtr;           // Receives new handle containing string data
```

Retrieves data from a string variable which may be local or global.

Returns the string data via a new handle and stores the handle in *hPtr. On return, if *hPtr is not NULL then you own the new handle and must dispose it.

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.

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.

See **VarName Parameters** on page 83 for further information.

Added for Igor Pro 6.10. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

Thread Safety

FetchStringDataUsingVarName is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

StoreStringDataUsingVarName

```
int
StoreStringDataUsingVarName(varName, buf, len)
const char* varName;    // Pointer to varName field in runtime param struct
const char* buf;        // String contents
BCInt 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.

See **VarName Parameters** on page 83 for further information.

Example

See `VDTReadBinary.cpp`.

Thread Safety

`StoreStringDataUsingVarName` is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetOperationWaveRef

```
int
SetOperationWaveRef (waveH, waveRefIdentifier)
waveHndl waveH;           // The destination wave you created.
int waveRefIdentifier;     // 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 (see **DataFolderAndName Parameters** on page 84) 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 `$dest-Wave`). 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 `waveRefIdentifier` 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
waveH = NULL;
err = MDMakeWave (&waveH, p->dest.name, p->dest.dfH, dimSizes, type, overwrite);
if (waveH != NULL) {
    int waveRefIdentifier = p->destParamsSet [0];
    err = SetOperationWaveRef (waveH, waveRefIdentifier);
}
```

See **DataFolderAndName Destination Wave Example** on page 85 for a more complete example of using a DataFolderAndName parameter for an operation that creates a destination wave.

Returns 0 or an error code.

Thread Safety

StoreStringDataUsingVarName is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetOperationWaveRef

```
int
GetOperationWaveRef(dfH, name, waveRefIdentifier, destWaveHPtr)
DataFolderHandle dfH;           // dfH field from DataFolderAndName structure
const char* name;               // name field from DataFolderAndName structure
int waveRefIdentifier;          // Value from destParamsSet field
waveHndl destWaveHPtr;         // Output wave reference is stored here
```

Returns via destWaveHPtr a reference to the destination wave if it was specified using a wave reference or NULL otherwise.

NOTE: Except in rare cases, you should not call this function. Use **GetOperationDestWave** instead.

The GetOperationWaveRef routine supports the /DEST flag for an operation.

If the destination wave was specified by passing a wave reference to /DEST, this routine returns the wave handle via destWaveHPtr.

If the destination wave was not specified via a wave reference, this routine returns NULL via destWaveHPtr and you must create the destination wave yourself.

The rule is: If a simple name or the name of a structure wave field are passed to /DEST, the specified wave is returned via destWaveHPtr. If \$<string> or a path are passed to /DEST, NULL is returned.

Here are typical cases where a wave reference specifies the destination wave:

```
SampleOp /DEST=jack
Wave w = root:jack
SampleOp /DEST=w
```

In these cases, GetOperationWaveRef would return a handle to wave jack via destWaveHPtr. In the first command, Operation Handler creates an automatic wave reference in the calling user-defined function if you are running Igor Pro 6.20 or later.

Here are typical cases where a wave reference is not used to specify the destination wave:

```
String name = "jack"
SampleOp /DEST=$name           // Dest specified by $<string>
SampleOp /DEST=root:jack       // Dest specified by path
SampleOp /DEST=:jack           // Dest specified by path
```

In these cases, there is no wave reference for the destination wave and GetOperationWaveRef would return NULL via destWaveHPtr.

The user can specify that a wave reference be created by appending /WAVE after the /DEST flag, like this:

```
String name = "jack"
SampleOp /DEST=$name/WAVE=wDest
SampleOp /DEST=root:jack/WAVE=wDest
SampleOp /DEST=:jack/WAVE=wDest
```

"/WAVE=wDest" is an "inline WAVE reference statement". An inline WAVE reference does not determine which wave is the destination wave but merely provides a place for the operation to return a reference to the destination wave.

If a wave reference for the destination wave exists, it must be set to refer to the destination wave once that wave has been determined or created. You set the wave reference by calling **SetOperationWaveRef** later in the process.

See **GetOperationDestWave** for an example.

Added for Igor Pro 6.20. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

Thread Safety

GetOperationWaveRef is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetOperationDestWave

```
int
GetOperationDestWave(dfH, name, destWaveRefIdentifier, options,
                    dimensionSizes, dataType, destWaveHPtr, destWaveCreatedPtr)
DataFolderHandle dfH;           // dfH field from DataFolderAndName structure
const char* name;               // name field from DataFolderAndName structure
int destWaveRefIdentifier;       // Value from destParamsSet field
int options;                     // Bit flags defined below
CountInt dimensionSizes[];      // Array of desired dest wave dimension sizes
int dataType;                   // Desired dest wave data type
waveHndl destWaveHPtr;          // Output: Wave reference is stored here
int* destWaveCreatedPtr;        // Output: Set to 1 if dest wave was created
```

Returns via destWaveHPtr a reference to the destination wave specified by the /DEST flag in an operation. GetOperationDestWave optionally creates the wave if it does not already exist. If the wave does exist, GetOperationDestWave changes it as specified by the options parameter.

GetOperationDestWave is typically called to support the /DEST flag for an operation. It determines which wave is to be the destination wave, optionally creates it if it does not already exist, optionally overwrites it or sets its dimensions and data type if it does already exist, and returns the wave handle via destWaveHPtr.

In Igor Pro 6.20 or later, GetOperationDestWave respects wave references passed as the destination wave when called from a user-defined function.

If GetOperationDestWave succeeds, it returns 0 as the function result and a non-NULL wave handle via destWaveHPtr. If it fails it returns a non-zero error code as the function result and NULL via destWaveHPtr. Possible errors include NOMEM, NO_TEXTNUMERIC_WAVE_OVERWRITE, and NAME_WAV_CONFLICT (the destination wave already exists and overwrite is not specified).

destWaveRefIdentifier is used by GetOperationDestWave to access a user-defined function local wave reference for the destination wave, if such a wave reference was specified with the /DEST flag. The example below shows how to obtain destWaveRefIdentifier from the operation's runtime parameter structure.

The options parameter is a bitwise combination of the following flags:

Flag	Description
kOpDestWaveOverwriteOK	Permits returning a pre-existing wave. If kOpDestWaveOverwriteOK is omitted and the wave specified by /DEST already exists, GetOperationDestWave returns an error.
kOpDestWaveChangeExistingWave	If the destination wave already exists its dimensions and data type are changed based on the dimensionSizes and dataType parameters but other properties (units, dimension scaling, dimension labels, wave note) are left unchanged.
kOpDestWaveOverwriteExistingWave	If the destination wave already exists it is overwritten using the specified dimension sizes and data type. The overwritten wave's properties (units, dimension scaling, dimension labels, wave note) are set to default values. Its data is left as is and you should treat it as uninitialized.
kOpDestWaveMakeFreeWave	Destination wave will be a new free wave. Requires Igor Pro 6.20 or later.
kOpDestWaveMustAlreadyExist	Destination wave must already exist. Otherwise an error is returned.

The choice between kOpDestWaveChangeExistingWave and kOpDestWaveOverwriteExistingWave must be made on a case-by-case basis. If you are converting an existing operation to use GetOperationDestWave you should choose between kOpDestWaveChangeExistingWave and kOpDestWaveOverwriteExistingWave so that the behavior of your operation will not change. For most new applications, kOpDestWaveOverwriteExistingWave is preferable since it makes the output of the operation independent of pre-existing conditions.

If kOpDestWaveChangeExistingWave and kOpDestWaveOverwriteExistingWave are both cleared, then GetOperationDestWave will not change the dimensions and data type of a pre-existing destination wave. In this case, it is up to you to make sure that the dimensions and data type of the returned wave are appropriate and change them if necessary.

If kOpDestWaveChangeExistingWave and kOpDestWaveOverwriteExistingWave are both set, this is a programmer error and an error is returned.

If your operation has a /FREE flag, signifying that the destination wave should be a free wave, set the kOpDestWaveMakeFreeWave bit. If kOpDestWaveMakeFreeWave is set, GetOperationDestWave makes a new free wave as the destination wave and returns it via destWaveHPtr and all other option flag are ignored. This requires Igor Pro 6.20 or later. See **Free Waves** on page 132 for further details on free waves.

If kOpDestWaveMakeFreeWave is not set, this procedure is followed: If the destination wave was specified via a wave reference by the calling user-defined function, GetOperationDestWave returns that wave via destWaveHPtr. Otherwise it looks for an existing wave specified by dfH and name. If there is an existing wave, GetOperationDestWave reuses it. Otherwise it creates a new wave using dfH and name. The rules for obtaining the destination wave via a wave reference are explained in the documentation for **GetOperationWaveRef**.

If kOpDestWaveMustAlreadyExist is set, then, if the destination wave does not already exist, the NOWAV ("expected wave name") error is returned. You may want to test for NOWAV and return a more specific error code. If you set kOpDestWaveMustAlreadyExist you must also set kOpDestWaveOverwriteOK because if kOpDestWaveOverwriteOK is cleared GetOperationDestWave returns an error if the destination wave exists.

If you want to return an existing wave only without overwriting or changing it, pass (kOpDestWaveMustAlreadyExist | kOpDestWaveOverwriteOK) and leave all other bits cleared.

dimensionSizes is an array of wave dimension sizes as shown in the example below. If kOpDestWaveMustAlreadyExist is set and kOpDestWaveChangeExistingWave, kOpDestWaveOverwriteExistingWave and kOpDestWaveMakeFreeWave are cleared then dimensionSizes is not used and you can pass NULL.

dataType is normally a standard Igor data type (e.g., NT_FP64, (NT_FP32 | NT_CMPLX), TEXT_WAVE_TYPE). However, if you pass NULL for dimensionSizes, you must pass -1 for dataType indicating that you do not want to change the data type.

If destWaveCreatedPtr is not NULL, *destWaveCreatedPtr is set to 1 if GetOperationDestWave created a new wave or to 0 if GetOperationDestWave reused an existing wave.

In the following example, the operation is defined by this Operation Handler syntax:

```
SampleOp /DEST=DataFolderAndName:{dest,real}
```

This results in an operation runtime parameter structure containing:

```
int DESTFlagEncountered;
DataFolderAndName dest;
int DESTFlagParamsSet[1];
```

Here is typical usage for an operation with a /DEST flag. In this example, p is a pointer to the operation runtime parameter structure:

```
waveHndl destWaveH;           // Destination wave handle.
int destWaveRefIdentifier;
char destWaveName[MAX_OBJ_NAME+1];
DataFolderHandle dfH;
int dataType;
CountInt dimensionSizes[MAX_DIMENSIONS+1];
int destWaveCreated;
int options;
int err;

destWaveH = NULL;
destWaveRefIdentifier = 0;

strcpy(destWaveName, "W_SampleOp"); // Default dest wave name
dfH = NULL;                         // Default is current data folder
dataType = NT_FP64;
MemClear(dimensionSizes, sizeof(dimensionSizes));
dimensionSizes[ROWS] = 100;
if (p->DESTFlagEncountered) {
    strcpy(destWaveName, p->dest.name);
    dfH = p->dest.dfH;
    // If a wave reference was used, p->destParamsSet[0] contains info that
    // GetOperationDestWave uses to find the address of the wave reference.
    destWaveRefIdentifier = p->DESTFlagParamsSet[0];
}

options = kOpDestWaveOverwriteOK | kOpDestWaveOverwriteExistingWave;
err = GetOperationDestWave(dfH, destWaveName, destWaveRefIdentifier,
    options, dimensionSizes, dataType, &destWaveH, &destWaveCreated);
if (err != 0)
    return err;

<Store output data in dest wave>
WaveHandleModified(destWaveH);

// Set wave reference to refer to destination wave.
if (destWaveRefIdentifier != 0)
    SetOperationWaveRef(destWaveH, destWaveRefIdentifier);
```

The call to `SetOperationWaveRef` sets the automatic destination wave reference created by Operation Handler when you use the `DataFolderAndName` syntax shown above in your operation template.

`GetOperationDestWave` was added for Igor Pro 6.20. If you call this with an earlier version of Igor, it will still return a destination wave but without recognizing any wave referenced passed via the `/DEST` flag. For example:

```
Wave w = root:wave0
SampleOp /O /DEST=w
```

In Igor Pro 6.20 and later, this overwrites a wave named `wave0` in the root data folder. However, prior to Igor Pro 6.20, it creates a wave named `w` in the current data folder. This is how destination waves worked in XOPs prior to Igor Pro 6.20.

If `kOpDestWaveMakeFreeWave` is set in the options parameter and you are running with a version of Igor prior to 6.20, `GetOperationDestWave` returns `IGOR_OBSOLETE`.

Thread Safety

`GetOperationDestWave` is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

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 (a `waveHndl`). You receive this handle as a parameter to an external operation or function or get it using the **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.

MakeWave

```
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
CountInt 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.

NOTE: For historical reasons, if `numPoints` is zero, **MakeWave** creates a 128-point wave. To create a zero-point wave, use **MDMakeWave**.

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.

Prior to XOP Toolkit 6, if the `numPoints` parameter was zero, the wave was created with 128 points. This behavior was a relic of a time when Igor required that a wave contain at least two points (prior to Igor Pro 3.0). Now when `numPoints` is zero, a zero-point wave is created.

`type` is one of the following:

Type	Wave Element Contains
NT_FP32	32 bit floating-point
NT_FP64	64 bit floating-point

Type	Wave Element Contains
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)
WAVE_TYPE	Wave handle or NULL Requires Igor Pro 6.10 or later
DATAFOLDER_TYPE	Data folder handle or NULL Requires Igor Pro 6.10 or later

To make a complex numeric wave, OR with NT_CMPLX, for example (NT_FP64 | NT_CMPLX). Any numeric wave can be complex. Text waves, wave reference waves and data folder reference waves can not be complex.

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.

NOTE: The block of memory referenced by the wave handle can be relocated if you increase the size of an existing wave. To avoid dangling pointer bugs, see **Resizing Handles** on page 199.

Thread Safety

MakeWave is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

ChangeWave

```
int
ChangeWave(waveHandle, numPoints, type)
waveHndl waveHandle;      // WaveHndl for wave to change
CountInt 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 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.

NOTE: The block of memory referenced by the wave handle can be relocated if you increase the size of the wave. To avoid dangling pointer bugs, see **Resizing Handles** on page 199.

Thread Safety

ChangeWave is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread

created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

KillWave

```
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 dependency formula. An XOP can also prevent a wave used in the main Igor thread from being killed by responding to the OBJINUSE message.

For information on a related technique useful in preemptive Igor threads, see **Wave Reference Counting** on page 128.

Thread Safety

KillWave is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

FetchWave

```
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, modern XOPs must use **FetchWaveFromDataFolder** instead of FetchWave.

Thread Safety

FetchWave is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

FetchWaveFromDataFolder

```
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**.

Thread Safety

FetchWaveFromDataFolder is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

WaveType

```
int
WaveType (waveHandle)
waveHndl waveHandle;          // Handle to wave's data structure
```

Returns the wave's data type.

The returned value is one of the following except that, if the wave is complex, the NT_CMPLX bit will be set:

Type	Wave Element Contains
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)
WAVE_TYPE	Wave handle or NULL Igor Pro 6.10 or later
DATAFOLDER_TYPE	Data folder handle or NULL Igor Pro 6.10 or later

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.

Thread Safety

WaveType is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

WavePoints

```
CountInt
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**.

Thread Safety

WavePoints is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

WaveModDate

TickCountInt

WaveModDate (waveHandle)

waveHndl waveHandle; // Handle to wave's data structure

Returns wave modification date. This is an unsigned 32-bit number 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.

Thread Safety

WaveModDate is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

WaveModCount

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 (waveH) ;
. . .
waveModCount2 = WaveModCount (waveH) ;
if (waveModCount2 != waveModCount1)
    // Wave has changed.
```

Thread Safety

WaveModCount is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

WaveModState

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.

Thread Safety

WaveModState is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

WaveLock

```
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.

Thread Safety

WaveLock is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetWaveLock

```
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.

Thread Safety

SetWaveLock is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

WaveScaling

```
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:

$X \text{ value} = x0 + dx * p$

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.

Thread Safety

WaveScaling is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetWaveScaling

```
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/P x” operation from within Igor. Igor calculates the X value for point p of the wave as:

$$X \text{ value} = x0 + dx * p$$

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.

Thread Safety

SetWaveScaling is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

WaveUnits

```
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.

Thread Safety

WaveUnits is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetWaveUnits

```
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.

Units can be up to 49 (MAX_UNIT_CHARS) characters long.

New XOPs should use **MDSetWaveUnits** instead of **SetWaveUnits** so that they can access units for higher dimensions.

Thread Safety

SetWaveUnits is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

WaveNote

```
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 remove the null terminator when you are done. See **Understand the Difference Between a String in a Handle and a C String** on page 237.

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.

Thread Safety

WaveNote is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetWaveNote

```
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 is 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.

Thread Safety

SetWaveNote is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

WaveName

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.

Thread Safety

WaveName is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

WaveData

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 resized. See **Dangling Pointers** on page 232 for details.

See Chapter 7 for detailed instructions on accessing wave data.

Thread Safety

WaveData is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

WaveMemorySize

void*

WaveMemorySize(waveH, which)

waveHndl waveH; // Wave handle

int which; // Code for what information you want

This routine was created for use by the Data Browser XOP. Most XOPs have no need to use it.

In Igor Pro 6 and before wave handles were regular handles. Although there was little reason to do so, you could call GetHandleSize on them and get a correct answer.

In Igor Pro 7, to support 64 bits on Macintosh, WaveMetrics had to change the way a wave's data was allocated. As a result, a wave handle is no longer a regular handle. Calling GetHandleSize on it will return the wrong result or crash.

This routine provides a way to get information about the memory used by a wave that will work in Igor Pro 7 as well as in earlier versions.

which is defined as follows:

Value	Information Returned
0	The size in bytes of the wave handle itself.
1	The size in bytes of the wave header (a structure located at the start of the wave handle).
2	The size in bytes of the wave numeric or text data.
3	The size in bytes of the wave handle and the numeric or text data.
4	In Igor Pro 7 or later, the total number of bytes required to store all wave data. This includes memory used to store wave properties such as the wave note, units and dimension labels. In Igor Pro 6 or before which=4 returns the same value as which=3 - the size in bytes of the wave handle and the numeric or text data.

which=2, 3 and 4 include the size of numeric data for numeric a wave and the size of text data for a text wave.

which=4 returns the most accurate assessment of the total memory usage of the wave. This is because, in Igor Pro 7 and later, it includes wave property data and also because, in some cases as explained below, the data for text waves is stored in separate objects, not in the wave handle itself.

In Igor Pro 7, the size of the wave handle itself may not accurately reflect the amount of memory required by a text wave. This is because, in most cases, Igor Pro 7 stores the text data for each element in a separate text element object and stores pointers to the text element objects in the wave handle itself. Thus the size of the wave handle itself reflects the storage needed for the pointers, not the storage needed for the actual text.

In Igor Pro 7 and later, determining the size of text data for a large text wave, using which=2, which=3 or which=4, can be time-consuming. This is because, in most cases, Igor Pro 7 stores the data for each element of the wave in a separate text element object and stores pointers to the text element objects in the wave handle itself. Thus Igor must determine the sum of the sizes of each of the objects.

Added in XOP Toolkit 6.40. Works with Igor Pro 3 or later.

Thread Safety

WaveMemorySize is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

CalcWaveRange

int*

CalcWaveRange (wrp)

WaveRangeRecPtr wrp;

Given coordinates expressed in point numbers or X values, CalcWaveRange returns the corresponding range of point numbers within the wave.

wrp is a pointer to a WaveRangeRec structure defined in IgorXOP.h.

See **Wave Range Parameters** on page 82 for an example.

Returns 0 if OK or a non-zero error code.

Thread Safety

CalcWaveRange is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread

created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

WaveHandleModified

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.

Thread Safety

WaveHandleModified is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

WaveModified

void

WaveModified(waveName)

char* waveName; // C string containing wave's name

NOTE: This routine is obsolete and may be removed in a future version of the XOP Toolkit. Use **WaveHandleModified** instead.

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.

Thread Safety

WaveModified is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDMakeWave

int

MDMakeWave(waveHPtr, waveName, dataFolderH, dimensionSizes, type, overwrite)

waveHndl* waveHPtr; // Place to put waveHndl for new wave

char* waveName; // C string containing name of Igor wave

DataFolderHandle dataFolderH; // Handle to data folder or NULL.

CountInt 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.

type is one of the following:

Type	Wave Element Contains
NT_FP32	32 bit floating-point
NT_FP64	64 bit floating-point

Type	Wave Element Contains
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)
WAVE_TYPE	Wave handle or NULL Requires Igor Pro 6.10 or later
DATAFOLDER_TYPE	Data folder handle or NULL Requires Igor Pro 6.10 or later

To make a complex numeric wave, OR with NT_CMPLX, for example (NT_FP64 | NT_CMPLX). Any numeric wave can be complex. Text waves, wave reference waves and data folder reference waves can not be complex.

If overwrite is non-zero, an existing wave with the same name will be overwritten.

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.

If you are running with Igor Pro 6.20 or later, you can pass -1 for dataFolderH to create a free wave. This would be appropriate if, for example, your external function were called from a user-defined function with a flag parameter indicating that the user wants to create a free wave. You must be sure that you are running with Igor Pro 6.20 or a later. If you are running with an earlier version, this will cause a crash. For example:

```
if (igorVersion < 620)
    return IGOR_OBSOLETE; // Can't create free wave in old Igor
result = MDMakeWave(&waveH, "freejack", (DataFolderHandle)-1,
                    dimensionSizes, type, overwrite);
```

When making a free wave, the overwrite parameter is irrelevant and is ignored.

You can also create a free wave using **GetOperationDestWave**.

See **Free Waves** on page 132 for further details on free waves.

Returns error code or 0 if wave was made.

NOTE: The block of memory referenced by the wave handle can be relocated if you increase the size of an existing wave. To avoid dangling pointer bugs, see **Resizing Handles** on page 199.

Example

```
waveHndl waveH;
char waveName[MAX_OBJ_NAME+1];
CountInt 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;
```

Thread Safety

MDMakeWave is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDGetWaveDimensions

```
int
MDGetWaveDimensions(waveH, numDimensionsPtr, dimensionSizes)
waveHndl waveH;           // Handle to the wave of interest
int* numDimensionsPtr;     // Number of dimensions in the wave
CountInt 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

```
int numDimensions;
CountInt dimensionSizes[MAX_DIMENSIONS+1];
int result;

if (result = MDGetWaveDimensions(waveH, &numDimensions, dimensionSizes))
    return result;
```

Thread Safety

MDGetWaveDimensions is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDChangeWave

```
int
MDChangeWave(waveH, dataType, dimensionSizes)
waveHndl waveH;           // Handle to the wave of interest
int dataType;             // New numeric type or -1 for no change
CountInt 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

For numeric waves, the data types may be ORed with NT_COMPLEX to make the wave complex. Text waves, wave reference waves and data folder reference waves can not be complex.

Igor can not convert a text wave to a numeric wave or vice-versa and you will receive an error if you attempt to do this.

dimensionSizes[i] contains the desired number of points for dimension i.

For a wave with 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.

NOTE: The block of memory referenced by the wave handle can be relocated if you increase the size of the wave. To avoid dangling pointer bugs, see **Resizing Handles** on page 199.

Example

```
int dataType;
CountInt 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(waveH, dataType, dimensionSizes))
    return result;
```

Thread Safety

MDChangeWave is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDChangeWave2

```
int
MDChangeWave2(waveH, dataType, dimensionSizes, mode)
waveHndl waveH;           // Handle to the wave of interest
int dataType;             // New numeric type or -1 for no change
CountInt 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. 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. 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.

NOTE: The block of memory referenced by the wave handle can be relocated if you increase the size of the wave. To avoid dangling pointer bugs, see **Resizing Handles** on page 199.

Thread Safety

MDChangeWave2 is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDGetWaveScaling

```
int
MDGetWaveScaling(waveH, dimension, sfAPtr, sfBPtr)
waveHndl waveH;           // 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 element e is:

$$\langle \text{scaled index} \rangle = \text{sfA}[i] * e + \text{sfB}[i]$$

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 126 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(waveH, ROWS, &sfA, &sfB))    // Get X scaling
    return result;
if (result = MDGetWaveScaling(waveH, COLUMNS, &sfA, &sfB)) // Get Y scaling
    return result;
```

Thread Safety

MDGetWaveScaling is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDSetWaveScaling

```
int
MDSetWaveScaling(waveH, dimension, sfAPtr, sfBPtr)
waveHndl waveH;           // 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 element e is:

$$\langle \text{scaled index} \rangle = \text{sfA}[i] * e + \text{sfB}[i]$$

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 126 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(waveH, ROWS, &sfA, &sfB))    // Set X scaling
    return result;
sfA = 1.0; sfB = 0.0;
if (result= MDSetWaveScaling(waveH, COLUMNS, &sfA, &sfB)) // Set Y scaling
    return result;
```

Thread Safety

MDSetWaveScaling is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDGetWaveUnits

```
int
MDGetWaveUnits(waveH, dimension, units)
waveHndl waveH;           // 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 126 for a discussion of the distinction between dimension units and data units.

Example

```
int numDimensions;
int dimension;
char units[MAX_UNIT_CHARS+1];
char buf[256];
int result;

if (result = MDGetWaveDimensions(waveH, &numDimensions, NULL))
    return result;
for (dimension=0; dimension<numDimensions; dimension++) {
    if (result = MDGetWaveUnits(waveH, dimension, units))
        return result;
    sprintf(buf, "Units for dimension %d: \"%s\"\\\"CR_STR, dimension, units);
    XOPNotice(buf);
}
```

Thread Safety

MDGetWaveUnits is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDSetWaveUnits

```
int
MDSetWaveUnits(waveH, dimension, units)
waveHndl waveH;           // 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 126 for a discussion of the distinction between dimension units and data units.

Example

```
char units[MAX_UNIT_CHARS+1];
int result;

if (result = MDSetWaveUnits(waveH, 0, "s")) // Set X units to seconds
    return result;
if (result = MDSetWaveUnits(waveH, -1, "v")) // Set data units to volts
    return result;
```

Thread Safety

MDSetWaveUnits is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDGetDimensionLabel

```
int
MDGetDimensionLabel(waveH, dimension, element, dimLabel)
waveHndl waveH;           // Handle to the wave of interest
int dimension;             // The dimension of interest.
IndexInt 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.

Thread Safety

MDGetDimensionLabel is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDSetDimensionLabel

```
int
MDSetDimensionLabel(waveH, dimension, element, dimLabel)
waveHndl waveH;           // Handle to the wave of interest
int dimension;             // The dimension of interest
IndexInt 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.

Thread Safety

MDSetDimensionLabel is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetWaveDimensionLabels

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.

Thread Safety

GetWaveDimensionLabels is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetWaveDimensionLabels

```
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.

Thread Safety

SetWaveDimensionLabels is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDAccessNumericWaveData

```
int
MDAccessNumericWaveData(waveH, accessMode, dataOffsetPtr)
waveHndl waveH;                // Handle to the wave of interest
int accessMode;
BCInt* 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 122 for a comparison of the various methods.

waveH 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 122 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.

```
int numDimensions;
CountInt dimensionSizes[MAX_DIMENSIONS+1];
CountInt numRows, numColumns;
IndexInt row, column;
BCInt dataOffset;
int type, type2, isComplex;
char* cp; short* sp; SInt32* SInt32p;
unsigned char* ucp; unsigned short* usp; UInt32* UInt32p;
float* fp; double* dp;
int result;

type = WaveType(waveH);
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(waveH, &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(waveH,kMDWaveAccessMode0,&dataOffset))
    return result;
dp= (double*)((char*)(*waveH) + dataOffset); fp=(float*)dp;
SInt32p= (SInt32*)dp; sp=(short*)dp; cp=(char*)dp;
UInt32p=(UInt32*)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:
                *SInt32p++ += 1; if (isComplex) *SInt32p++ += 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:
                *UInt32p++ += 1; if (isComplex) *UInt32p++ += 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.
                return NT_FNOT_AVAIL;    // Func not supported on this type.
        }
    }
}
```

Thread Safety

`MDAccessNumericWaveData` is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it

from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDGetNumericWavePointValue

```
int
MDGetNumericWavePointValue(waveH, indices, value)
waveHndl waveH;           // Handle to the wave of interest
IndexInt 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 for **MDSetNumericWavePointValue**.

Thread Safety

MDGetNumericWavePointValue is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDSetNumericWavePointValue

```
int
MDSetNumericWavePointValue(waveH, indices, value)
waveHndl waveH;           // Handle to the wave of interest
IndexInt indices[MAX_DIMENSIONS]; // Identifies the point of interest
double value[2];          // Value passed 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 use 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.

```
int numDimensions;
CountInt dimensionSizes[MAX_DIMENSIONS+1];
CountInt numRows, numColumns;
IndexInt row, column;
IndexInt indices[MAX_DIMENSIONS];
int isComplex;
double value[2];
int result;

isComplex = WaveType(waveH) & NT_CMPLX;

if (result = MDGetWaveDimensions(waveH, &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(waveH, indices, value))
            return result;
        value[0] += 1; // Real part
        if (isComplex)
            value[1] += 1; // Imag part
        if (result = MDSetNumericWavePointValue(waveH, indices, value))
            return result;
    }
}
```

Thread Safety

MDSetNumericWavePointValue is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDGetDPDataFromNumericWave

```
int
MDGetDPDataFromNumericWave(waveH, dPtr)
waveHndl waveH;           // 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(waveH)*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 for MDStoreDPDataInNumericWave.

Thread Safety

MDGetDPDataFromNumericWave is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDStoreDPDataInNumericWave

```
int
MDStoreDPDataInNumericWave(waveH, dPtr)
waveHndl waveH;           // 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 MDGetNumericWavePointValue.

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.

```
int numDimensions;
CountInt dimensionSizes[MAX_DIMENSIONS+1];
CountInt numRows, numColumns;
BCInt numBytes;
IndexInt row, column;
int isComplex;
double* dPtr;
double* dp;
int result;

isComplex = WaveType(waveH) & NT_CMPLX;

if (result = MDGetWaveDimensions(waveH, &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(waveH) * sizeof(double); // Bytes needed for copy
if (isComplex)
    numBytes *= 2;
dPtr = (double*)NewPtr(numBytes);
if (dPtr==NULL)
    return NOMEM;

if (result = MDGetDPDataFromNumericWave(waveH, 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(waveH, dPtr)) {
    DisposePtr((Ptr)dPtr);
    return result;
}

DisposePtr((Ptr)dPtr);
```

Thread Safety

MDStoreDPDataInNumericWave is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

FetchNumericValue

```
int
FetchNumericValue(type, dataStartPtr, index, value)
int type;                // Igor numeric data type
char* dataStartPtr;      // Pointer to start of wave data
IndexInt 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.

The function result is zero or an error code.

Thread Safety

FetchNumericValue is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

StoreNumericValue

```
int
StoreNumericValue(type, dataStartPtr, index, value)
int type;                // Igor numeric data type
char* dataStartPtr;      // Pointer to start of wave data
IndexInt 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.

The function result is zero or an error code.

Thread Safety

StoreNumericValue is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDGetTextWavePointValue

```
int
MDGetTextWavePointValue(waveH, indices, textH)
waveHndl waveH;           // Handle to the wave of interest
IndexInt 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. If you pass the handle back to Igor, you must remove the null terminator. See **Understand the Difference Between a String in a Handle and a C String** on page 237.

See the example below for MDSetTextWavePointValue.

Thread Safety

MDGetTextWavePointValue is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MDSetTextWavePointValue

```
int
MDSetTextWavePointValue(waveH, indices, textH)
waveHndl waveH;           // Handle to the wave of interest
IndexInt 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 added a null terminator to 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**. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

```
int numDimensions;
IndexInt dimensionSizes[MAX_DIMENSIONS+1];
CountInt numRows, numColumns;
IndexInt row, column;
IndexInt indices[MAX_DIMENSIONS];
Handle textH;
int result;

if (result = MDGetWaveDimensions(waveH, &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(waveH, indices, textH))
            break;
        if (PtrAndHand("*", textH, 1)) { // Append an asterisk to handle
            result = NOMEM;
            break;
        }
        if (result = MDSetTextWavePointValue(waveH, indices, textH))
            break;
    }
}

DisposeHandle(textH);
```

Thread Safety

MDSetTextWavePointValue is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetTextWaveData

```
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	<p>The returned handle contains one null-terminated C string for each element of the wave.</p> <p>Example:</p> <pre>"Zero"<null> "One"<null> "Two"<null></pre>

Mode	Data Format
Mode=1	<p>The returned handle contains a list of 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.</p> <p>The offsets are 32 bits in IGOR32 and 64 bits in IGOR64.</p> <p>The text for each element in the wave is represented by a null-terminated C string.</p> <p>Example:</p> <pre> <Offset to "Zero"> <Offset to "One"> <Offset to "Two"> <Extra offset> "Zero"<null> "One"<null> "Two"<null> </pre>
Mode=2	<p>The returned handle contains a list of offsets to strings followed by the string data.</p> <p>The offsets are 32 bits in IGOR32 and 64 bits in IGOR64.</p> <p>The text for each element in the wave is not null-terminated.</p> <p>Example:</p> <pre> <Offset to "Zero"> <Offset to "One"> <Offset to "Two"> <Extra offset> "Zero" "One" "Two" </pre>

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.

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`.

Thread Safety

`GetTextWaveData` is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetTextWaveData

```
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. The offsets are 32 bits in IGOR32 and 64 bits in IGOR64.

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.

In Igor Pro 6.23, the SetTextWaveData routine was changed to return an error if the wave is locked or if the wave was passed to a thread as a parameter to the thread worker function. This change is in Igor, not in the XOP Toolkit.

Thread Safety

SetTextWaveData is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

HoldWave

```
int
HoldWave(waveH)
waveHndl waveH;
```

waveH is a wave handle that you have obtained from Igor.

As of XOP Toolkit 6.30, if waveH is NULL, HoldWave does nothing and returns 0.

HoldWave tells Igor that you are holding a handle referencing the wave and that it should not be deleted until you release it by calling **ReleaseWave**. This prevents a crash that would occur if Igor deleted a wave to which you held a handle and you later used that handle.

In most XOP programming, such as a straight-forward external operation or external function, you do not need to call HoldWave/ReleaseWave. If you are just using the wave handle temporarily during the execution of your external function or operation and you make no calls that could potentially delete the wave then you do not need to and should not call HoldWave and ReleaseWave.

You need to call HoldWave/ReleaseWave only if you are storing a wave handle over a period during which Igor could delete the wave. For example, you might indefinitely store a wave handle in a global variable or you might do a callback, such as **XOPCommand** or **CallFunction**, that could cause the wave to be deleted. In such cases, call HoldWave to prevent Igor from deleting the wave until you are finished with it. Then call ReleaseWave to permit Igor to delete the wave.

HoldWave returns 0 or IGOR_OBSOLETE as the function result.

See **Wave Reference Counting** on page 128 for background information and further details.

HoldWave returns an error code as the function result, typically 0 for success or IGOR_OBSOLETE.

HoldWave was added in Igor Pro 6.20. If you call this with an earlier version of Igor, HoldWave will return IGOR_OBSOLETE and do nothing.

Prior to XOP Toolkit 6.30, HoldWave had a second parameter. That parameter was removed in XOP Toolkit 6.30. This change does not affect XOPs compiled with earlier versions of the XOP Toolkit. It will cause a compile error that you will have to correct if you used HoldWave and compile with XOP Toolkit 6.30 or later.

Thread Safety

HoldWave is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

ReleaseWave

```
int  
ReleaseWave(waveRefPtr)  
waveHndl* waveRefPtr;
```

Tells Igor that you are no longer holding a wave.

waveRefPtr contains the address of your waveHndl variable that refers to a wave. ReleaseWave sets *waveRefPtr to NULL so your waveHndl variable is not valid after you call ReleaseWave.

If *waveRefPtr is NULL on input then ReleaseWave does nothing and returns 0.

See **Wave Reference Counting** on page 128 for details.

ReleaseWave returns an error code as the function result, typically 0 for success or IGOR_OBSOLETE.

ReleaseWave was added in Igor Pro 6.20. If you call this with an earlier version of Igor, ReleaseWave will return IGOR_OBSOLETE and do nothing.

Thread Safety

ReleaseWave is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 169. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

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 to access global variables referenced by NVAR and SVAR variables in user-defined functions.

In addition to the routines in this section, you can use **GetDataFolderObject** and **SetDataFolderObject** to access global variables.

These routines do not permit you to access local variables in user-defined functions.

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. Some of 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 under **Operation Handler Routines** on page 246.

To set file loader output variables, see **Routines for File-Loader XOPs** on page 353.

Variable

```
int
Variable(varName, varType)
char* varName;           // C string containing name of variable
int varType;             // Data type of variable
```

NOTE: We recommend that you use **SetIgorIntVar**, **SetIgorFloatingVar**, **SetIgorComplexVar** or **SetIgorStringVar** instead of **Variable** as they are easier to use.

When called from the command line or from a user-defined function, **Variable** creates a global numeric or string variable in the current data folder.

When called from a macro, if the **VAR_GLOBAL** bit is set in the **varType** parameter, **Variable** creates a global numeric or string variable in the current data folder. If **VAR_GLOBAL** is cleared **Variable** creates a macro-local variable. To give consistent behavior it is recommended that you always set the **VAR_GLOBAL** bit.

You can not create a local variable in a user-defined function except through **Operation Handler** (see page 72).

When your XOP is called from the command line or from a user-defined function, the **varType** parameter is defined as follows:

varType	Action
0 or VAR_GLOBAL	Creates global string variable
NT_FP64 or (NT_FP64 VAR_GLOBAL)	Creates global numeric variable
(NT_FP64 NT_CMPLX) or (NT_FP64 NT_CMPLX VAR_GLOBAL)	Creates global complex numeric variable

If you omit **VAR_GLOBAL** and you were called from a macro then a macro-local variable is created.

The function result is 0 if **Variable** was able to create the variable or an Igor error code if not.

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 72) to implement your external operation. Operation Handler will create output variables for you so you should not use Variable for that purpose.

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);
}
```

Thread Safety

Variable is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

FetchNumVar

```
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.

When called from the command line or from a user-defined function, FetchNumVar looks for a global variable in the current data folder.

When called from a macro, FetchNumVar first looks for a macro-local variable with the specified name. If none is found, it looks for a global variable in the current data folder.

You can not fetch a local variable in a user-defined function except through **Operation Handler** (see page 72).

Returns -1 if the variable does not exist or the numeric type of the variable if it does.

The returned 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.

Thread Safety

FetchNumVar is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

StoreNumVar

```
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
```

NOTE: We recommend that you use **SetIgorIntVar**, **SetIgorFloatingVar** or **SetIgorComplexVar** instead of **StoreNumVar** as they are easier to use.

Sets the value of the named Igor variable.

When called from the command line or from a user-defined function, **StoreNumVar** looks for a global variable in the current data folder.

When called from a macro, **StoreNumVar** first looks for a macro-local variable with the specified name. If none is found, it looks for a global variable in the current data folder.

You can not set a local variable in a user-defined function except through **Operation Handler** (see page 72).

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.

Thread Safety

StoreNumVar is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

FetchStrVar

```
int
FetchStrVar(varName, stringPtr)
char* varName;           // C string containing name of string
char stringPtr[256];      // Receives contents of string
```

Fetches the contents of the named Igor string variable.

When called from the command line or from a user-defined function, **FetchStrVar** looks for a global variable in the current data folder.

When called from a macro, **FetchStrVar** first looks for a macro-local variable with the specified name. If none is found, it looks for a global variable in the current data folder.

You can not fetch a local variable in a user-defined function except through **Operation Handler** (see page 72).

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 must point to an array of 256 characters.

On output, **stringPtr** contains a null-terminated C string.

FetchStrVar will never return more than 255 characters. If you want to access any characters beyond the first 255, use **FetchStrHandle** instead of **FetchStrVar**.

Thread Safety

FetchStrVar is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

StoreStrVar

```
int
StoreStrVar(varName, stringPtr)
char* varName;           // C string containing name of string
char* stringPtr;         // C string to store in string variable
```

NOTE: We recommend that you use **SetIgorStringVar** instead of StoreStrVar as it is easier to use.

Sets the contents of the named Igor string variable.

When called from the command line or from a user-defined function, StoreStrVar looks for a global variable in the current data folder.

When called from a macro, StoreStrVar first looks for a macro-local variable with the specified name. If none is found, it looks for a global variable in the current data folder.

You can not set a local variable in a user-defined function except through **Operation Handler** (see page 72).

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.

stringPtr is a null-terminated C string.

There is no limit to the length of the C string pointed to by stringPtr.

Thread Safety

StoreStrVar is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

FetchStrHandle

```
Handle
FetchStrHandle(varName)
char* varName;           // C string containing name of string
```

Returns the handle containing the text for the named Igor string variable.

Returns NULL if there is no such string variable.

When called from the command line or from a user-defined function, FetchStrHandle looks for a global variable in the current data folder.

When called from a macro, FetchStrHandle first looks for a macro-local variable with the specified name. If none is found, it looks for a global variable in the current data folder.

You can not fetch a local variable in a user-defined function except through **Operation Handler** (see page 72).

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. You must remove the null terminator before control returns to Igor. See **Understand the Difference Between a String in a Handle and a C String** on page 237.

NOTE: The returned handle belongs to Igor. You must not dispose of or otherwise modify this handle.

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.

Thread Safety

FetchStrHandle is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetIgorIntVar

```
int
SetIgorIntVar(numVarName, value, forceGlobal)
char* numVarName;      // Name of the Igor numeric variable to set
int value;             // Value to set it to
int forceGlobal;       // Non-zero to create a global variable
```

Creates an Igor numeric variable if it does not already exist. Sets the variable to an integer value.

When called from the command line or from a user-defined function, SetIgorIntVar always creates and/or sets a global variable in the current data folder.

When called from a macro, SetIgorIntVar creates and/or sets a global variable in the current data folder if forceGlobal is non-zero and creates and/or sets a macro-local variable if forceGlobal is 0. To get consistent behavior regardless of how you are called, pass 1 for forceGlobal.

You can not set a local variable in a user-defined function except through **Operation Handler** (see page 72).

The function result is 0 or an error code.

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 72) to implement your external operation. Operation Handler will create output variables for you so you should not use SetIgorIntVar for that purpose.

Thread Safety

SetIgorIntVar is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetIgorFloatingVar

```
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
```

Creates an Igor numeric variable if it does not already exist. Sets the variable to a floating-point value.

When called from the command line or from a user-defined function, SetIgorFloatingVar always creates and/or sets a global variable in the current data folder.

When called from a macro, SetIgorFloatingVar creates and/or sets a global variable in the current data folder if forceGlobal is non-zero and creates and/or sets a macro-local variable if forceGlobal is 0. To get consistent behavior regardless of how you are called, pass 1 for forceGlobal.

You can not set a local variable in a user-defined function except through **Operation Handler** (see page 72).

The function result is 0 or an error code.

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 72) to implement your external operation. Operation Handler will create output variables for you so you should not use SetIgorFloatingVar for that purpose.

Thread Safety

SetIgorFloatingVar is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetIgorComplexVar

```
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
```

Creates an Igor complex numeric variable if it does not already exist. Sets the variable to a complex floating-point value.

When called from the command line or from a user-defined function, SetIgorComplexVar always creates and/or sets a global variable in the current data folder.

When called from a macro, SetIgorComplexVar creates and/or sets a global variable in the current data folder if forceGlobal is non-zero and creates and/or sets a macro-local variable if forceGlobal is 0. To get consistent behavior regardless of how you are called, pass 1 for forceGlobal.

You can not set a local variable in a user-defined function except through **Operation Handler** (see page 72).

The function result is 0 or an error code.

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 72) to implement your external operation. Operation Handler will create output variables for you so you should not use SetIgorComplexVar for that purpose.

Thread Safety

SetIgorComplexVar is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetIgorStringVar

```
int
SetIgorStringVar(stringVarName, stringVarValue, forceGlobal)
char* stringVarName;   // Name of the Igor string variable to set
char* stringVarValue;  // C string value to store in string variable
int forceGlobal;       // Non-zero to create a global string
```

Creates an Igor string variable if it does not already exist. Stores the specified string in the variable.

When called from the command line or from a user-defined function, SetIgorStringVar always creates and/or sets a global variable in the current data folder.

When called from a macro, SetIgorStringVar creates and/or sets a global variable in the current data folder if forceGlobal is non-zero and creates and/or sets a macro-local variable if forceGlobal is 0. To get consistent behavior regardless of how you are called, pass 1 for forceGlobal.

You can not set a local variable in a user-defined function except through **Operation Handler** (see page 72).

stringVarValue is a null-terminated C string of any length.

The function result is 0 or an error code.

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 72) to implement your external operation. Operation Handler will create output variables for you so you should not use SetIgorStringVar for that purpose.

Thread Safety

SetIgorStringVar is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetNVAR

```
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 to an external operation or external function.

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.

See also **NVARs and SVARs In Structures** on page 212.

Thread Safety

GetNVAR is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetNVAR

```
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 to an external operation or external function.

*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.

See also **NVARs and SVARs In Structures** on page 212.

Thread Safety

SetNVAR is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetSVAR

```
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.

syp is a pointer to an SVAR field in an Igor structure. The Igor structure pointer would be passed into the XOP as a parameter to an external operation or external function.

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. You must test for this case before using *strHPtr.

NOTE: *strHPtr belongs to Igor. Do not dispose it or alter it. Use this function only to read the contents of the string.

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. You must remove the null terminator before control returns to Igor. See **Understand the Difference Between a String in a Handle and a C String** on page 237.

Returns 0 or an error code.

See also **NVARs and SVARs In Structures** on page 212.

Thread Safety

GetSVAR is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetSVAR

```
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.

syp is a pointer to an SVAR field in an Igor structure. The Igor structure pointer would be passed into the XOP as a parameter to an external operation or external function.

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.

Note that the text in the handle is *not* null terminated. See **Understand the Difference Between a String in a Handle and a C String** on page 237.

Returns 0 or an error code.

See also **NVARs and SVARs In Structures** on page 212.

Thread Safety

SetSVAR is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

Routines for Accessing Data Folders

Data folders provide hierarchical data storage for Igor data. 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 138 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 via a `DataFolderAndName` operation parameter or by calling one of these routines:

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.

GetDataFolderNameOrPath

```
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, `dataFolderPathOrName` 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, `GetDataFolderNameOrPath` 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 `dataFolderPathOrName`.

The function result is 0 or error code.

Thread Safety

`GetDataFolderNameOrPath` is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetDataFolderIDNumber

```
int
GetDataFolderIDNumber(dataFolderH, IDNumberPtr)
DataFolderHandle dataFolderH;
int* IDNumberPtr;
```

Returns the unique ID number for the data folder via `IDNumberPtr`.

If `dataFolderH` is NULL, `GetDataFolderIDNumber` 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.

Thread Safety

GetDataFolderIDNumber is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetDataFolderProperties

```
int
GetDataFolderProperties(dataFolderH, propertiesPtr)
DataFolderHandle dataFolderH;
int* 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.

Thread Safety

GetDataFolderProperties is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetDataFolderProperties

```
int
SetDataFolderProperties(dataFolderH, properties)
DataFolderHandle dataFolderH;
int properties;
```

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.

Thread Safety

SetDataFolderProperties is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetDataFolderListing

```
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 237.

If dataFolderH is NULL, GetDataFolderListing 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(0, &dataFolderH))
    return result;
if (result = GetDataFolderListing(dataFolderH, 15, &listingH))
    return result;
<Use contents of listingH>
DisposeHandle(listingH);
```

Thread Safety

GetDataFolderListing is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetRootDataFolder

```
int
GetRootDataFolder(refNum, rootDataFolderHPtr)
int 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.

Thread Safety

GetRootDataFolder is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetCurrentDataFolder

```
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.

Thread Safety

GetCurrentDataFolder is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetCurrentDataFolder

```
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.

Thread Safety

SetCurrentDataFolder is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetNamedDataFolder

```
int
GetNamedDataFolder(startingDataFolderH, dataFolderPath, dataFolderHPtr)
DataFolderHandle startingDataFolderH;
char dataFolderPath[MAXCMDLEN+1];
DataFolderHandle* dataFolderHPtr;
```

Returns in *dataFolderHPtr the data folder specified by startingDataFolderH and dataFolderPath or NULL if there is no such data folder.

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:".

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.

Absolute and relative paths may include the trailing colon or not, except for the case of root.

Passing "root" as dataFolderPath will not find the root data folder because "root" is a data folder name, not a path. 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 = GetRootDataFolder(0, &rootH))
    return result;
if (result = GetNamedDataFolder(rootH, ":Packages:", &dataFolderH))
    return result;
```

Thread Safety

GetNamedDataFolder is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetDataFolderByIDNumber

```
int
GetDataFolderByIDNumber(IDNumber, dataFolderHPtr)
int IDNumber;
DataFolderHandle* dataFolderHPtr;
```

Returns via *dataFolderHPtr the data folder handle associated with the specified ID number or NULL if there is no such data folder.

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.

The function result is 0 or error code.

Thread Safety

GetDataFolderByIDNumber is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetParentDataFolder

```
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, GetParentDataFolder uses the current data folder.

Passing the root data folder as dataFolderH is an error. In this case GetParentDataFolder returns NO_PARENT_DATAFOLDER.

If dataFolderH references a free data folder then GetParentDataFolder returns NO_PARENT_DATAFOLDER.

The function result is 0 or error code.

Example

```
// This example shows how to determine if a data folder is free
int dataFolderIsFree = 0;           // Assume not free
if (dfH != GetRootDataFolder()) {
    if (GetParentDataFolder(dfH) == NO_PARENT_DATAFOLDER)
        dataFolderIsFree = 1;      // It's free
}
```

Thread Safety

GetParentDataFolder is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetNumChildDataFolders

```
int
GetNumChildDataFolders(parentDataFolderH, numChildDataFolderPtr)
DataFolderHandle parentFolderH;
int* numChildDataFolderPtr;
```

Returns the number of child data folders in the specified parent data folder.

If parentDataFolderH is NULL, GetNumChildDataFolders uses the current data folder.

The function result is 0 or error code.

Thread Safety

GetNumChildDataFolders is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetIndexedChildDataFolder

```
int
GetIndexedChildDataFolder(parentDataFolderH, index, childDataFolderHPtr)
DataFolderHandle parentFolderH;
int 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, GetIndexedChildDataFolder uses the current data folder.

The function result is 0 or error code.

Thread Safety

GetIndexedChildDataFolder is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetWavesDataFolder

```
int
GetWavesDataFolder(waveH, dataFolderHPtr)
waveHndl waveH;
DataFolderHandle* dataFolderHPtr;
```

Returns via dataFolderHPtr the handle to the data folder containing the specified wave. This will be NULL if the wave is a free wave.

Data folder handles belong to Igor so you should not modify or dispose them.

The function result is 0 or error code.

Thread Safety

GetWavesDataFolder is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

NewDataFolder

```
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.

Thread Safety

NewDataFolder is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

KillDataFolder

```
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 or used in dependency formulas).

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 must not use it again for any purpose.

The function result is 0 or error code.

Thread Safety

KillDataFolder is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

DuplicateDataFolder

```
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.

Thread Safety

DuplicateDataFolder is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MoveDataFolder

```
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.

Thread Safety

MoveDataFolder is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

RenameDataFolder

```
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.

Thread Safety

RenameDataFolder is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetNumDataFolderObjects

```
int
GetNumDataFolderObjects(dataFolderH, objectType, numObjectsPtr)
DataFolderHandle dataFolderH;
int objectType;
int* numObjectsPtr;
```

Returns via numObjectsPtr the number of objects of the specified type in the specified data folder.

If dataFolderH is NULL, GetNumDataFolderObjects uses the current data folder.

objectType is one of the following:

WAVE_OBJECT	Waves
VAR_OBJECT	Numeric variables
STR_OBJECT	String variables

To find the number of data folders within the data folder, use **GetNumChildDataFolders**.

Thread Safety

GetNumDataFolderObjects is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetIndexedDataFolderObject

```
int
GetIndexedDataFolderObject (dataFolderH, objectType, index, objectName, vp)
DataFolderHandle dataFolderH;
int objectType;
int 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, GetIndexedDataFolderObject uses the current data folder.

objectType is one of the following:

WAVE_OBJECT	Waves
VAR_OBJECT	Numeric variables
STR_OBJECT	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->waveH 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.

Prior to XOP Toolkit 6 the waveH field of the DataObjectValue structure was named wavH.

The handles returned via the waveH and strH fields belong to Igor. Do not modify or dispose them.

The function result is 0 or error code.

Thread Safety

GetIndexedDataFolderObject is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetDataFolderObject

```
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, GetDataFolderObject 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->waveH field to wave's handle.
VAR_OBJECT	Stores numeric variable's value in vp->nv field.
STR_OBJECT	Sets vp->strH field to string variable's handle.
DATAFOLDER_OBJECT	Sets vp->dfH field to data folder's handle.

Prior to XOP Toolkit 6 the waveH field of the DataObjectValue structure was named wavH.

The handles returned via the waveH, strH and dfH fields belong to Igor. Do not modify or dispose them.

Note that the text in a string 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. You must remove the null terminator before control returns to Igor. See **Understand the Difference Between a String in a Handle and a C String** on page 237.

The function result is 0 or error code if there is no object with the specified name.

Example

```
DataObjectValue v;
int objectType;
waveHndl waveH;
Handle strH;
double numVarValue;
DataFolderHandle dfH;
int err;

if (err = GetDataFolderObject(dfH, "AnObject", &objectType, &v))
    return err;
switch(objectType) {
    case WAVE_OBJECT:
        waveH = v.waveH;
        <Do something with waveH>;
```

```

        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;
}

```

Thread Safety

GetDataFolderObject is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetDataFolderObject

```

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, SetDataFolderObject uses the current data folder.

objectType must be one of the following:

WAVE_OBJECT	Wave
VAR_OBJECT	Numeric variable
STR_OBJECT	String variable
DATAFOLDER_OBJECT	Data folder

vp is a pointer to a DataObjectValue union, defined in IgorXOP.h. The action of SetDataFolderObject 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).

Note that the text in a string handle is *not* null terminated. Use **SetHandleSize** to set the number of bytes in the handle. To store a C string in a handle, use **PutCStringInHandle**. See **Understand the Difference Between a String in a Handle and a C String** on page 237.

The function result is 0 or error code if there is no object with the specified name.

Example

```
DataObjectValue v;
int objectType;
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;
```

Thread Safety

SetDataFolderObject is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

KillDataFolderObject

```
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, KillDataFolderObject uses the current data folder.

objectType is one of the following:

WAVE_OBJECT	Wave
VAR_OBJECT	Numeric variable
STR_OBJECT	String variable

DATAFOLDER_OBJECT Data folder

NOTE: If you attempt to kill a wave that is in use (e.g., in a graph, table or dependency formula) 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.

Thread Safety

KillDataFolderObject is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

DuplicateDataFolderObject

```
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, DuplicateDataFolderObject uses the current data folder.

objectType is one of the following:

WAVE_OBJECT	Wave
VAR_OBJECT	Numeric variable
STR_OBJECT	String variable
DATAFOLDER_OBJECT	Data folder

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 **CheckName**, **CleanupName** and **UniqueName2** or the higher-level **CreateValidDataObjectName** routine.

The function result is 0 or error code.

Thread Safety

DuplicateDataFolderObject is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

MoveDataFolderObject

```
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, MoveDataFolderObject uses the current data folder.

If destDataFolderH is NULL, MoveDataFolderObject uses the current data folder.

objectType is one of the following:

WAVE_OBJECT	Wave
VAR_OBJECT	Numeric variable
STR_OBJECT	String variable
DATAFOLDER_OBJECT	Data folder

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.

Thread Safety

MoveDataFolderObject is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

RenameDataFolderObject

```
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, RenameDataFolderObject uses the current data folder.

objectType is one of the following:

WAVE_OBJECT	Wave
VAR_OBJECT	Numeric variable
STR_OBJECT	String variable
DATAFOLDER_OBJECT	Data folder

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.

NOTE: Prior to Igor Pro 6.23 this function returned an error if objectType was DATAFOLDER_OBJECT. If you want to rename a data folder in Igor Pro 6.22 or earlier you must use **RenameDataFolder**.

The function result is 0 or error code.

Thread Safety

RenameDataFolderObject is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

HoldDataFolder

```
int
HoldDataFolder (dfH)
DataFolderHandle dfH;
```

Tells Igor that you are holding a reference to a data folder and that the data folder should not be deleted until you call ReleaseDataFolder.

dfH is a data folder handle that you have obtained from Igor.

As of XOP Toolkit 6.30, if dfH is NULL, HoldDataFolder does nothing and returns 0.

See **Data Folder Reference Counting** on page 140 for details.

HoldDataFolder returns an error code as the function result, typically 0 for success or IGOR_OBSOLETE.

HoldDataFolder was added in Igor Pro 6.20. If you call this with an earlier version of Igor, HoldDataFolder will return IGOR_OBSOLETE and do nothing.

Prior to XOP Toolkit 6.30, HoldDataFolder had a second parameter. That parameter was removed in XOP Toolkit 6.30. This change does not affect XOPs compiled with earlier versions of the XOP Toolkit. It will cause a compile error that you will have to correct if you used HoldDataFolder and compile with XOP Toolkit 6.30 or later.

Thread Safety

HoldDataFolder is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

ReleaseDataFolder

```
int
ReleaseDataFolder (dfRefPtr)
DataFolderHandle* dfRefPtr;
```

Tells Igor that you are no longer holding a data folder.

dfRefPtr contains the address of your DataFolderHandle variable that refers to a data folder. ReleaseDataFolder sets *dfRefPtr to NULL so your DataFolderHandle variable is not valid after you call ReleaseDataFolder.

If *dfRefPtr is NULL on input then ReleaseDataFolder does nothing and returns 0.

See **Data Folder Reference Counting** on page 140 for details.

ReleaseDataFolder returns an error code as the function result, typically 0 for success or IGOR_OBSOLETE.

ReleaseDataFolder was added in Igor Pro 6.20. If you call this with an earlier version of Igor, ReleaseDataFolder will return IGOR_OBSOLETE and do nothing.

Thread Safety

ReleaseDataFolder is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

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.

Most of these routines were added in XOP Toolkit 6.40 to replace ancient routines that were based on the Macintosh Menu Manager. Although these routines were added in XOP Toolkit 6.40, they work with all versions of Igor with minor exceptions.

If you are updating an existing XOP that uses one of the old routines, you will need to change your XOP to use the new routines instead. See “XOP Menu Compatibility” in the XOPMenus.c file for information that will help you do the update.

Routines for dealing with dialog popup menus are described under **Dialog Popup Menus** on page 332.

ResourceToActualMenuID

```
int  
ResourceToActualMenuID(resourceMenuID)  
int resourceMenuID;
```

Given the ID of a MENU resource in the XOP’s resource fork, ResourceToActualMenuID returns the actual menu ID of that resource in memory.

ResourceToActualMenuID returns 0 if the XOP did not add this menu to Igor.

See **Determining Which Menu Was Chosen** on page 148 for a discussion of resource IDs versus actual IDs.

Thread Safety

ResourceToActualMenuID is not thread-safe.

ActualToResourceMenuID

```
int  
ActualToResourceMenuID(menuID)  
int menuID;
```

Given the ID of a menu in memory, ActualToResourceMenuID returns the resource ID of the MENU resource in the XOP’s resource fork.

ActualToResourceMenuID returns 0 if the XOP did not add this menu to Igor.

See **Determining Which Menu Was Chosen** on page 148 for a discussion of resource IDs versus actual IDs.

Thread Safety

ActualToResourceMenuID is not thread-safe.

ResourceToActualItem

```
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, ResourceToActualItem returns the actual item number of that item in the Igor menu.

Both menu item specification numbers and menu item numbers start from one.

ResourceToActualItem returns 0 if the XOP did not add this menu item to Igor.

See **Determining Which Menu Was Chosen** on page 148 for a discussion of resource items versus actual items.

Thread Safety

ResourceToActualItem is not thread-safe.

ActualToResourceItem

```
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, ActualToResourceItem 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.

ActualToResourceItem returns 0 if the XOP did not add this menu item to Igor.

See **Determining Which Menu Was Chosen** on page 148 for a discussion of resource items versus actual items.

Thread Safety

ActualToResourceItem is not thread-safe.

SetIgorMenuItem

```
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
int param;             // Normally not used and should be 0
```

Enables or disables the built-in Igor menu item associated with message.

Call SetIgorMenuItem only if your XOP window is the active window. You would call it in response to a MENUENABLE message from Igor.

For example, if the XOP wants to enable the Copy menu item, it would call Igor as follows:

```
if (IsXOPWindowActive((xopWindowRef)) // Is XOP window active?
    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:

```
if (IsXOPWindowActive((xopWindowRef)) // Is XOP window active?
    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.

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.

Thread Safety

SetIgorMenuItem is not thread-safe.

XOPActualMenuIDToMenuRef

XOPMenuRef

XOPActualMenuIDToMenuRef(int actualMenuID)

int actualMenuID; // The menu ID of a menu that your XOP added to Igor

Returns the menu reference for the specified menu.

Use XOPActualMenuIDToMenuRef to get an menu reference for an Igor menu. This is needed to enable or disable an XOP menu item in an Igor menu.

If you are trying to get a menu reference for a menu added to Igor by your XOP use **XOPResourceMenuIDToMenuRef** instead of XOPActualMenuIDToMenuRef.

Always check the returned menuRef for NULL before using it.

NOTE: XOPActualMenuIDToMenuRef returns NULL if actualMenuID is not valid.

NOTE: When running with Igor Pro 6 XOPActualMenuIDToMenuRef returns NULL for main menubar menus that are hidden (e.g., via **XOPHideMainMenu**). With Igor Pro 7 it returns the menu reference whether the menu is hidden or not.

XOPResourceMenuIDToMenuRef returns your menu reference whether it is hidden or not.

See MenuXOP1.cpp for an example.

Thread Safety

XOPActualMenuIDToMenuRef is not thread-safe.

XOPResourceMenuIDToMenuRef

XOPMenuRef

XOPResourceMenuIDToMenuRef(int resourceMenuID)

int resourceMenuID;

Given the ID of a MENU resource in the XOP's resource fork, XOPResourceMenuIDToMenuRef returns a reference to the menu. You can use this reference with other XOP menu routines to modify the menu.

Always check the returned menu reference for NULL before using it.

NOTE: XOPResourceMenuIDToMenuRef returns NULL if XOP did not add this menu.

Unlike **XOPActualMenuIDToMenuRef**, **XOPResourceMenuIDToMenuRef** returns the menu reference even if it is a hidden main menubar menu.

Thread Safety

XOPResourceMenuIDToMenuRef is not thread-safe.

XOPGetMenuInfo

```
int
XOPGetMenuInfo(menuRef, actualMenuID, menuTitle, isVisible, reserved1, reserved2)
XOPMenuRef menuRef;
int* actualMenuID;           // Output
char* menuTitle;             // Output
int* isVisible;              // Output
void* reserved1;             // Reserved for future use - pass NULL
void* reserved2;             // Reserved for future use - pass NULL
```

Returns information about the menu specified by menuRef.

*actualMenuID is set to the actual menu ID of the menu. You can pass NULL if you don't care about this value.

menuTitle is a pointer to a 256 byte array of chars. You can pass NULL if you don't care about this value. This feature requires Igor Pro 6.40 or later. When running with earlier versions *menuTitle is set to 0.

*isVisible is set to the truth that the menu is visible in the menu bar. This returns 0 for hidden main menubar menus and 1 for visible main menubar menus. You can pass NULL if you don't care about this value. This feature requires Igor Pro 6.40 or later. When running with earlier versions *isVisible is set to -1.

reserved1 and reserved2 are reserved for future use. Pass NULL for these parameters.

The function result is 0 for success or an error code.

See MenuXOP1.cpp for an example.

Added in Igor Pro 6.40. When running with earlier versions only the actualMenuID parameter is supported.

Thread Safety

XOPGetMenuInfo is not thread-safe.

XOPCountMenuItems

```
int
XOPCountMenuItems(menuRef)
XOPMenuRef menuRef;
```

Returns the number of items in the menu.

See MenuXOP1.cpp for an example.

Added in Igor Pro 6.40 but works with any version.

Thread Safety

XOPCountMenuItems is not thread-safe.

XOPShowMainMenu

```
int
XOPShowMainMenu(menuRef, beforeMenuID)
XOPMenuRef menuRef;
int beforeMenuID;           // The menu ID of a menu in the menubar or 0
```

menuRef is a reference to an XOP main menubar menu. XOPShowMainMenu makes the menu appear in the menubar if it was previously hidden.

beforeMenuID is the actual menu ID of a menu in the menubar or 0 to show the specified menu at the end of the menubar. In most cases you should pass 0.

Returns 0 for success or an error code.

See MenuXOP1.cpp for an example.

Added in Igor Pro 6.40 but works with any version.

Thread Safety

XOPShowMainMenu is not thread-safe.

XOPHideMainMenu

```
int
XOPHideMainMenu(menuRef)
XOPMenuRef menuRef;
```

menuRef is a reference to an XOP main menubar menu. XOPHideMainMenu removes the menu from the menubar if it was previously showing.

Returns 0 for success or an error code.

See MenuXOP1.cpp for an example.

Added in Igor Pro 6.40 but works with any version.

Thread Safety

XOPHideMainMenu is not thread-safe.

XOPGetMenuItemInfo

```
int
XOPGetMenuItemInfo(menuRef, itemNumber, enabled, checked, reserved1, reserved2)
XOPMenuRef menuRef;
int itemNumber;           // A 1-based item number
int* enabled;             // Output - set to 1 if item is enabled, 0 otherwise
int* checked;             // Output - set to 1 if item is checked, 0 otherwise
void* reserved1;          // Reserved for future use - pass NULL
void* reserved2;          // Reserved for future use - pass NULL
```

XOPGetMenuItemInfo returns information about the menu item specified by menuRef and itemNumber.

menuRef is a reference to an XOP menu.

itemNumber is a 1-based item number.

*enabled is set to the truth that the menu item is enabled. You can pass NULL if you don't want to know if the menu item is enabled.

*checked is set to the truth that the menu item is checked. You can pass NULL if you don't want to know if the menu item is checked.

reserved1 and reserved2 are reserved for future use. Pass NULL for these parameters.

Returns 0 for success or an error code.

See MenuXOP1.cpp for an example.

Added in Igor Pro 6.40.

This function requires Igor Pro 6.40 or later. With earlier versions it returns NOT_IMPLEMENTED and *enabled and *checked are set to -1.

Thread Safety

XOPGetMenuItemInfo is not thread-safe.

XOPGetMenuItemText

```
int
XOPGetMenuItemText(menuRef, itemNumber, text)
XOPMenuRef menuRef;
int itemNumber;           // A 1-based item number
char text[256];           // Item text is returned here
```

XOPGetMenuItemText returns the text from an XOP menu item specified by menuRef and itemNumber.

menuRef is a reference to an XOP menu.

itemNumber is a 1-based item number.

text must be able to hold 255 characters plus the null terminator.

Returns 0 for success or an error code.

See MenuXOP1.cpp for an example.

Added in Igor Pro 6.40 but works with any version.

Thread Safety

XOPGetMenuItemText is not thread-safe.

XOPSetMenuItemText

```
int
XOPSetMenuItemText(menuRef, itemNumber, text)
XOPMenuRef menuRef;
int itemNumber;           // A 1-based item number
const char* text;         // The text for the item
```

XOPSetMenuItemText sets the text of an XOP menu item specified by menuRef and itemNumber.

menuRef is a reference to an XOP menu.

itemNumber is a 1-based item number.

text is C string (null-terminated) of 255 characters or less.

Returns 0 for success or an error code.

See MenuXOP1.cpp for an example.

Added in Igor Pro 6.40 but works with any version.

Thread Safety

XOPSetMenuItemText is not thread-safe.

XOPAppendMenuItem

```
int  
XOPAppendMenuItem(menuRef, text)  
XOPMenuRef menuRef;  
const char* text;          // The text for the item
```

XOPAppendMenuItem adds a menu item to the end of the XOP menu item specified by menuRef.

menuRef is a reference to an XOP menu.

text is C string (null-terminated) of 255 characters or less.

Returns 0 for success or an error code.

See MenuXOP1.cpp for an example.

Added in Igor Pro 6.40 but works with any version.

Thread Safety

XOPAppendMenuItem is not thread-safe.

XOPInsertMenuItem

```
int  
XOPInsertMenuItem(menuRef, afterItemNumber, text)  
XOPMenuRef menuRef;  
int afterItemNumber;       // A 1-based item number  
const char* text;          // The text for the item
```

XOPInsertMenuItem inserts a menu item in the XOP menu specified by menuRef.

menuRef is a reference to an XOP menu.

afterItemNumber is a 1-based item number. Pass 0 to insert the menu item at the beginning of the menu.

Pass n to insert the item after existing item n of the menu.

text is C string (null-terminated) of 255 characters or less.

Returns 0 for success or an error code.

See MenuXOP1.cpp for an example.

Added in Igor Pro 6.40 but works with any version.

Thread Safety

XOPInsertMenuItem is not thread-safe.

XOPDeleteMenuItem

```
int  
XOPDeleteMenuItem(xxx, xxx)  
XOPMenuRef menuRef;  
int itemNumber;            // A 1-based item number
```

XOPDeleteMenuItem removes the menu item specified by menuRef and itemNumber from an XOP menu.

menuRef is a reference to an XOP menu.

itemNumber is a 1-based item number.

Returns 0 for success or an error code.

See MenuXOP1.cpp for an example.

Added in Igor Pro 6.40 but works with any version.

Thread Safety

XOPDeleteMenuItem is not thread-safe.

XOPDeleteMenuItemRange

```
int
XOPDeleteMenuItemRange(menuRef, firstMenuItemNumber, lastMenuItemNumber)
XOPMenuRef menuRef;
int firstMenuItemNumber;    // A 1-based item number
int lastMenuItemNumber;    // A 1-based item number
```

XOPDeleteMenuItemRange removes a range of menu items from an XOP menu.

menuRef is a reference to an XOP menu.

firstMenuItemNumber and lastMenuItemNumber are 1-based item numbers. They are clipped to the range of valid item numbers so you can pass 1 for firstMenuItemNumber and 10000 for lastMenuItemNumber to delete all items from the menu.

Returns 0 for success or an error code.

See MenuXOP1.cpp for an example.

Added in Igor Pro 6.40 but works with any version.

Thread Safety

XOPDeleteMenuItemRange is not thread-safe.

XOPEnableMenuItem

```
int
XOPEnableMenuItem(menuRef, itemNumber)
XOPMenuRef menuRef;
int itemNumber;           // A 1-based item number
```

XOPEnableMenuItem enables the specified item.

menuRef is a reference to an XOP menu or an Igor menu.

itemNumber is a 1-based item number.

Returns 0 for success or an error code.

See MenuXOP1.cpp for an example.

Added in Igor Pro 6.40 but works with any version.

Thread Safety

XOPEnableMenuItem is not thread-safe.

XOPDisableMenuItem

```
int
XOPDisableMenuItem(menuRef, itemNumber)
XOPMenuRef menuRef;
int itemNumber;           // A 1-based item number
```

XOPDisableMenuItem disables the specified item.

menuRef is a reference to an XOP menu or an Igor menu.

itemNumber is a 1-based item number.

Returns 0 for success or an error code.

See MenuXOP1.cpp for an example.

Added in Igor Pro 6.40 but works with any version.

Thread Safety

XOPDisableMenuItem is not thread-safe.

XOPCheckMenuItem

```
int
XOPCheckMenuItem(menuRef, itemNumber, state)
XOPMenuRef menuRef;
int itemNumber;           // A 1-based item number
int state;                // 1 for check, 0 for uncheck
```

XOPCheckMenuItem adds or removes a checkmark from a menu item.

menuRef is a reference to an XOP menu or an Igor menu.

itemNumber is a 1-based item number.

Returns 0 for success or an error code.

See MenuXOP1.cpp for an example.

Added in Igor Pro 6.40 but works with any version.

Thread Safety

XOPCheckMenuItem is not thread-safe.

XOPFillMenu

```
void
XOPFillMenu(menuRef, afterItemNumber, itemList)
XOPMenuRef menuRef;       // A reference to an XOP menu
int afterItemNumber;       // A 1-based item number of an item in the menu
const char* itemList;     // Semicolon separated list of items to add to menu
```

itemList is C string (null-terminated) containing a semicolon separated list of items to be put into the menu.

afterItemNumber specifies where the items in itemList are to appear in the menu.

afterItemNumber = 0	New items appear at beginning of menu
afterItemNumber = 10000	New items appear at end of menu
afterItemNumber = item number	New items appear after specified existing item number

XOPFillMenu 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. Use XOPFillMenuNoMeta if you don't want this behavior.

Returns 0 or an error code.

Do not call XOPFillMenu for dialog popup menus. Use **FillPopupMenu** instead.

Thread Safety

XOPFillMenu is not thread-safe.

XOPFillMenuNoMeta

void

XOPFillMenuNoMeta(menuRef, afterItemNumber, itemList)

XOPMenuRef menuRef; // A reference to an XOP menu
int afterItemNumber; // A 1-based item number of an item in the menu
const char* itemList; // Semicolon separated list of items to add to menu

XOPFillMenuNoMeta works exactly like XOPFillMenu except that it does not honor meta-characters.

itemList is C string (null-terminated) containing a semicolon separated list of items to be put into the menu.

afterItemNumber specifies where the items in itemList are to appear in the menu.

afterItemNumber = 0	New items appear at beginning of menu
afterItemNumber = 10000	New items appear at end of menu
afterItemNumber = item number	New items appear after specified existing item number

Returns 0 or an error code.

Do not call XOPFillMenuNoMeta for dialog popup menu item. Use **FillPopupMenu** instead.

Thread Safety

XOPFillMenuNoMeta is not thread-safe.

XOPFillWaveMenu

int

XOPFillWaveMenu(menuRef, match, options, afterItemNumber)

XOPMenuRef menuRef; // A reference to an XOP menu
char* match; // "*" for all waves or match pattern
char* options; // Options for further selection of wave
int afterItemNumber; // A 1-based item 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 afterItemNumber or, if afterItemNumber is 0, at the beginning of the menu.

The meaning of the match, and options parameters is the same as for the built-in Igor WaveList function.

XOPFillWaveMenu does not treat any characters as meta-characters.

Do not call XOPFillWaveMenu for dialog popup menus. Use **FillWavePopupMenu** instead.

Returns 0 or an error code.

Thread Safety

XOPFillWaveMenu is not thread-safe.

XOPFillPathMenu

int

XOPFillPathMenu(menuRef, match, options, afterItemNumber)

XOPMenuRef menuRef; // A reference to an XOP menu
char* match; // "*" for all paths or match pattern
char* options; // Options for further selection of path
int afterItemNumber; // A 1-based item 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 `afterItemNumber` or, if `afterItemNumber` is 0, at the beginning of the menu.

The meaning of the `match` parameter is the same as for the built-in Igor PathList function.

options must be "".

Returns 0 or an error code.

XOPFillPathMenu does not treat any characters as meta-characters.

Do not call XOPFillPathMenu for dialog popup menus. Use **FillPathPopupMenu** instead.

Thread Safety

XOPFillPathMenu is not thread-safe.

XOPFillWinMenu

```
int
XOPFillWinMenu(menuRef, match, options, afterItemNumber)
XOPMenuRef menuRef;      // A reference to an XOP menu
char* match;             // "*" for all windows or match pattern
char* options;           // Options for further selection of windows
int afterItemNumber;     // A 1-based item 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 `afterItemNumber` or, if `afterItemNumber` is 0, at the beginning of the menu.

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.

Returns 0 or an error code.

XOPFillWinMenu does not treat any characters as meta-characters.

Do not call XOPFillWinMenu for dialog popup menus. Use **FillWindowPopupMenu** instead.

Thread Safety

XOPFillWinMenu is not thread-safe.

Routines for XOPs that Have Dialogs

These utilities assist an XOP in handling a standard modal dialog. See the sample file `VDTDialog.cpp` for an example of a modal dialog. See `GBLoadWaveDialog.cpp` for an example of an Igor-style modal dialog. See **Adding Dialogs** on page 203 for an overview.

GetXOPDialog

```
DialogPtr
GetXOPDialog(dialogID)
int dialogID;
```

GetXOPDialog 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`.

Thread Safety

GetXOPDialog is not thread-safe.

ShowDialogWindow

```
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.

Thread Safety

ShowDialogWindow is not thread-safe.

SetDialogPort

```
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.

Thread Safety

SetDialogPort is not thread-safe.

DoXOPDialog

```
void
DoXOPDialog(itemPtr)
short* itemPtr;
```

DoXOPDialog 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.cpp for an example.

Thread Safety

DoXOPDialog is not thread-safe.

XOPDialog

```
void
XOPDialog(filterProc, itemPtr)
ModalFilterUPP filterProc;
short* itemPtr;
```

XOPDialog 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.

Thread Safety

XOPDialog is not thread-safe.

DisposeXOPDialog

```
void
DisposeXOPDialog(theDialog)
DialogPtr theDialog;
```

DisposeXOPDialog 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.

Thread Safety

DisposeXOPDialog is not thread-safe.

SetDialogBalloonHelpID

```
void
SetDialogBalloonHelpID(balloonHelpID)
int balloonHelpID;          // ID of the balloon help resource for the dialog
```

NOTE: SetDialogBalloonHelpID is obsolete as Mac OS X does not support balloon help. See **Macintosh Balloon Help** on page 220 for details.

On the Macintosh, SetDialogBalloonHelpID 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, SetDialogBalloonHelpID does nothing.

Thread Safety

SetDialogBalloonHelpID is not thread-safe.

GetDBox

```
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, GetDBox 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.

Thread Safety

GetDBox is not thread-safe.

GetRadBut

```
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, GetRadBut returns zero if the radio button is not turned on, non-zero if it is turned on.

Thread Safety

GetRadBut is not thread-safe.

SetRadBut

```
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, SetRadBut turns that radio button on and others in the group off.

Thread Safety

SetRadBut is not thread-safe.

ToggleCheckBox

```
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, ToggleCheckBox toggles the state of the checkbox. Returns the new state (0 = off, 1 = on).

Thread Safety

ToggleCheckBox is not thread-safe.

GetCheckBox

```
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, GetCheckBox returns the state of the checkbox (0 = off, 1 = on).

Thread Safety

GetCheckBox is not thread-safe.

SetCheckBox

```
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), SetCheckBox sets the state of the checkbox.

Thread Safety

SetCheckBox is not thread-safe.

HiliteDControl

```
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 HiliteDControl sets its highlighting to reflect its state.

The control is enabled if enable is non-zero, disabled and grayed out otherwise.

Thread Safety

HiliteDControl is not thread-safe.

EnableDControl

```
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, EnableDControl enables the control and sets its highlighting to reflect its enabled state.

Thread Safety

EnableDControl is not thread-safe.

DisableDControl

```
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, DisableDControl disables the control and sets its highlighting to reflect its disabled state (grays the control out).

Thread Safety

DisableDControl is not thread-safe.

GetDText

```
int
GetDText(theDialog, itemNumber, theText)
XOP_DIALOG_REF theDialog; // DialogPtr on Mac, HWND on Windows
int itemNumber;
char theText[256]; // 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, GetDText returns the text in the item via theText.

The string returned is a null-terminated C string of up to 255 characters. theText must be big enough to hold 256 bytes.

The function result is the number of characters in the string returned via theText.

Thread Safety

GetDText is not thread-safe.

SetDText

```
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, SetDText sets the text in the item to the contents of theText.

Thread Safety

SetDText is not thread-safe.

GetDInt

```
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, GetDInt 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.

Thread Safety

GetDInt is not thread-safe.

SetDInt

```
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, SetDInt sets the text in the item to the number in theInt.

Thread Safety

SetDInt is not thread-safe.

GetDInt64

```
int
GetDInt64(theDialog, itemNumber, valPtr)
XOP_DIALOG_REF theDialog; // DialogPtr on Mac, HWND on Windows
int itemNumber;
SInt64* valPtr;           // Pointer to SInt64 to receive number in item
```

Given a dialog and the item number of an edit text item in that dialog, GetDInt64 returns the number entered in the item as a 64-bit signed integer via **valPtr**.

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.

Thread Safety

GetDInt64 is not thread-safe.

SetDInt64

```
void
SetDInt64(theDialog, itemNumber, val)
XOP_DIALOG_REF theDialog; // DialogPtr on Mac, HWND on Windows
int itemNumber;
SInt64 val;                // The number to put in edit text item
```

Given a dialog and the item number of an edit text item in that dialog, SetDInt64 sets the text in the item to the number specified by val.

Thread Safety

SetDInt64 is not thread-safe.

GetDDouble

```
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, GetDDouble 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.

Thread Safety

GetDDouble is not thread-safe.

SetDDouble

```
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, SetDDouble sets the text in the item to the number pointed to by theDouble.

Thread Safety

SetDDouble is not thread-safe.

SelEditItem

```
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 was 0, SelEditItem selected the text in the current edit text item. This is no longer supported.

If the dialog has no edit items, SelEditItem does nothing.

SelEditItem 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.

Thread Safety

SelEditItem is not thread-safe.

SelMacEditItem

```
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, SelMacEditItem 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.

Thread Safety

SelMacEditItem is not thread-safe.

DisplayDialogCmd

```
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 GBLoad-WaveDialog.cpp 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.

Thread Safety

DisplayDialogCmd is not thread-safe.

FinishDialogCmd

```
void
FinishDialogCmd(cmd, mode)
char* cmd; // C string containing command
int mode;
```

You call FinishDialogCmd at end of an Igor-style dialog.

cmd is a null-terminated 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.

Thread Safety

FinishDialogCmd is not thread-safe.

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 207.

InitPopMenus

```
void
InitPopMenus(theDialog)
XOP_DIALOG_REF theDialog; // DialogPtr on Mac, HWND on Windows
```

Call InitPopMenus 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.

Thread Safety

InitPopMenus is not thread-safe.

CreatePopupMenu

void

CreatePopupMenu(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, CreatePopupMenu does not treat any characters as meta-characters.

Thread Safety

CreatePopupMenu is not thread-safe.

GetPopupMenuHandle

MenuHandle

GetPopupMenuHandle(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.

GetPopupMenuHandle 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 GetPopupMenuHandle, 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.

Thread Safety

GetPopupMenuHandle is not thread-safe.

ItemIsPopupMenu

int

ItemIsPopupMenu(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.

ItemIsPopupMenu is used by the HandleItemHit routine in the GBLoadWaveDialog.cpp and VDTDia-log.cpp 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.

Thread Safety

ItemIsPopupMenu is not thread-safe.

AddPopupMenuItems

```
int
AddPopupMenuItems(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.

AddPopupMenuItems can be called only after the popup menu item has been initialized by calling Create-PopMenu.

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, AddPopupMenuItems does not treat any characters as meta-characters.

Thread Safety

AddPopupMenuItems is not thread-safe.

SetPopMatch

```
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.

Thread Safety

SetPopMatch is not thread-safe.

SetPopItem

```
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.

Thread Safety

SetPopItem is not thread-safe.

GetPopupMenu

void

GetPopupMenu(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.

GetPopupMenu can be called only after the popup menu item has been initialized by calling **CreatePopupMenu**.

If you are not interested in the text of the item selected, pass NULL for selStr.

Thread Safety

GetPopupMenu is not thread-safe.

DeletePopMenuItems

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.

DeletePopMenuItems can be called only after the popup menu item has been initialized by calling **CreatePopupMenu**.

Item numbers start from one. Pass 0 to delete all items.

Thread Safety

DeletePopMenuItems is not thread-safe.

FillPopupMenu

void

FillPopupMenu(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
int 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.

FillPopupMenu can be called only after the popup menu item has been initialized by calling **CreatePopupMenu**.

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, FillPopupMenu does not treat any characters as meta-characters.

Thread Safety

FillPopupMenu is not thread-safe.

FillWavePopupMenu

```
int
FillWavePopupMenu(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.

FillWavePopupMenu can be called only after the popup menu item has been initialized by calling **CreatePopupMenu**.

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.

FillWavePopupMenu 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, FillWavePopupMenu does not treat any characters as meta-characters.

Thread Safety

FillWavePopupMenu is not thread-safe.

FillPathPopupMenu

```
int
FillPathPopupMenu(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.

FillPathPopupMenu can be called only after the popup menu item has been initialized by calling **CreatePopupMenu**.

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.

FillPathPopupMenu 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, FillPathPopupMenu does not treat any characters as meta-characters.

Thread Safety

FillPathPopupMenu is not thread-safe.

FillWindowPopupMenu

```
int
FillWindowPopupMenu(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.

FillWindowPopupMenu can be called only after the popup menu item has been initialized by calling **CreatePopupMenu**.

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, FillWindowPopupMenu does not treat any characters as meta-characters.

Thread Safety

FillWindowPopupMenu is not thread-safe.

KillPopMenus

```
void
KillPopMenus(theDialog)
XOP_DIALOG_REF theDialog; // DialogPtr on Mac, HWND on Windows
```

Cleans up and disposes popup menu handles.

Call KillPopMenus just before disposing a dialog that uses popup menus.

Thread Safety

KillPopMenus is not thread-safe.

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 201.

Some of these routines use full paths to identify files. The full paths must be “native”.

On Macintosh, “native” means “HFS” (Hierarchical File System) which means that the path separator is a colon. Do not use POSIX paths (with forward slashes) because the Carbon routines called by the XOPSupport routines do not support forward slashes.

On Windows, “native” means that the path separator is a backslash.

MacToWinPath

```
int
MacToWinPath(path)
char path[MAX_PATH_LEN+1];
```

MacToWinPath handles file path format conversion for cross-platform XOPs. See **File Path Conversions** on page 202 for background information.

MacToWinPath converts a Macintosh HFS path into a Windows path by replacing 'X:' with 'X:\' 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, the volume name is "\\server\share".

If the path is already a Windows path, MacToWinPath does nothing.

The following characters are illegal in Windows file names and are considered illegal by Igor. For cross-platform compatibility, you should not use them:

\ / : * ? " < > | <All control characters>

Control characters are characters whose ASCII codes are less than 32 decimal (20 hexadecimal).

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. MacToWinPath 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, MacToWinPath 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.

Thread Safety

MacToWinPath is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

WinToMacPath

```
int
WinToMacPath(path)
char path[MAX_PATH_LEN+1];
```

WinToMacPath handles file path format conversion for cross-platform XOPs. See **File Path Conversions** on page 202 for background information.

WinToMacPath converts a Windows path into a Macintosh HFS 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 the path is already a Macintosh HFS path, MacToWinPath does nothing.

The following characters are illegal in Windows file names and are considered illegal by Igor. For cross-platform compatibility, you should not use them:

```
\ / : * ? " < > | <All control characters>
```

Control characters are characters whose ASCII codes are less than 32 decimal (20 hexadecimal).

NOTE: The path may be shorter on output than in was on input (':' or ':' changed to ':').

When running on an Asian language system, WinToMacPath 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.

Thread Safety

WinToMacPath is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

HFSToPosixPath

```
int
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.

On Mac OS X, POSIX paths use UTF-8 encoding.

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, HFSToPosixPath 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.

Thread Safety

HFSToPosixPath is thread-safe. It can be called from any thread.

GetNativePath

```
int
GetNativePath(filePathIn, filePathOut)
char filePathIn[MAX_PATH_LEN+1];
char filePathOut[MAX_PATH_LEN+1];
```

GetNativePath handles file path format conversion for cross-platform XOPs. See **File Path Conversions** on page 202 for background information.

GetNativePath copies filePathIn to filePathOut and then calls **WinToMacPath** when running on Macintosh and **MacToWinPath** when running on Windows. If filePathIn does not use the conventions of the current platform, it converts filePathOut to use those conventions.

filePathOut can point to the same memory as filePathIn or it can point to different memory.

When running on an Asian language system, GetNativePath 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.

Thread Safety

GetNativePath is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

ConcatenatePaths

```
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 202 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, ConcatenatePaths 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.

Thread Safety

ConcatenatePaths is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetDirectoryAndFileNameFromFullPath

```
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 HFS 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.

Thread Safety

GetDirectoryAndFileNameFromFullPath is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

FullPathPointsToFile

```
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, FullPathPointsToFile is double-byte-character-aware and assumes that the system default character encoding governs the path.

Thread Safety

FullPathPointsToFile is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

FullPathPointsToFolder

```
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, FullPathPointsToFolder is double-byte-character-aware and assumes that the system default character encoding governs the path.

Thread Safety

FullPathPointsToFolder is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetLeafName

```
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.

GetLeafName returns via leafName the leaf part of the path if it is a path or the contents of filePath if it is not a path.

Returns 0 if OK or an error code.

Thread Safety

GetLeafName is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

ParseFilePath

```
int  
ParseFilePath(mode, pathIn, separator, whichEnd, whichElement, pathOut)  
int mode;  
const char* pathIn;  
const char* separator;  
int whichEnd, whichElement;  
char pathOut[MAX_PATH_LEN+1];
```

The ParseFilePath function provides the ability to manipulate file paths and to extract sections of file paths.

This XOPSupport routine works the same as the Igor built-in ParseFilePath function.

For further explanation, see the documentation for ParseFilePath in the Igor Reference help file.

The output is returned via the pathOut parameter which must be able to hold MAX_PATH_LEN+1 bytes.

The function result is 0 for success or an Igor error code.

Added in Igor Pro 6.20B03. If you call this when running with an earlier version, it will return IGOR_OBSOLETE.

Thread Safety

ParseFilePath is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SpecialDirPath

```
int  
SpecialDirPath(pathID, domain, flags, createDir, pathOut)  
const char* pathID;  
int domain, flags, createDir;  
char pathOut[MAX_PATH_LEN+1];
```

The SpecialDirPath function returns a full path to a file system directory specified by pathID and domain. It provides a programmer with a way to access directories of special interest, such as the preferences directory, the Igor Pro User Files directory and the desktop directory.

This XOPSupport routine works the same as the Igor built-in SpecialDirPath function.

For further explanation, see the documentation for SpecialDirPath in the Igor Reference help file.

The output is returned via the pathOut parameter which must be able to hold MAX_PATH_LEN+1 bytes.

The function result is 0 for success or an Igor error code.

Added in Igor Pro 6.20B03. If you call this when running with an earlier version, it will return IGOR_OBSOLETE.

Thread Safety

SpecialDirPath is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

XOPCreateFile

```
int
XOPCreateFile(fullFilePath, overwrite, macCreator, macFileType)
const char* fullFilePath;
int overwrite;
int macCreator;
int macFileType;
```

Creates a file with the location and name specified by fullFilePath.

fullFilePath must be an HFS path (using colon separators) on Macintosh and a Windows path (using 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.

Thread Safety

XOPCreateFile is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

XOPDeleteFile

```
int
XOPDeleteFile(fullFilePath)
const char* fullFilePath;
```

Deletes the specified file.

fullFilePath must be an HFS path (using colon separators) on Macintosh and a Windows path (using backslashes) on Windows.

Returns 0 if OK or an error code.

Thread Safety

XOPDeleteFile is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

XOPOpenFile

```
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 an HFS path (using colon separators) on Macintosh and a Windows path (using backslashes) on Windows.

The function result is 0 if OK or an error code.

Thread Safety

XOPOpenFile is thread-safe. It can be called from any thread.

XOPCloseFile

```
int
XOPCloseFile(XOP_FILE_REF fileRef)
XOP_FILE_REF fileRef;           // Reference returned by XOPOpenFile
```

Closes the referenced file.

XOPCloseFile returns 0 if OK or an error code.

Thread Safety

XOPCloseFile is thread-safe. It can be called from any thread.

XOPReadFile

```
int
XOPReadFile(fileRef, count, buffer, numBytesReadPtr)
XOP_FILE_REF fileRef;           // Reference returned by XOPOpenFile
UInt32 count;                   // Count of bytes to read
void* buffer;                   // Where bytes are stored
UInt32* numBytesReadPtr;        // Output: number of bytes read
```

Reads count bytes from the referenced file into the buffer.

If numBytesReadPtr is not NULL, XOPReadFile 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.

Thread Safety

XOPReadFile is thread-safe. It can be called from any thread.

XOPReadFile2

```
int
XOPReadFile2(fileRef, count, buffer, numBytesReadPtr)
XOP_FILE_REF fileRef;           // Reference returned by XOPOpenFile
UInt32 count;                   // Count of bytes to read
void* buffer;                   // Where bytes are stored
UInt32* numBytesReadPtr;        // Output: number of bytes read
```

Reads count bytes from the referenced file into the buffer.

If numBytesReadPtr is not NULL, XOPReadFile2 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.

Thread Safety

XOPReadFile2 is thread-safe. It can be called from any thread.

XOPReadFile64

```
int
XOPReadFile64(fileRef, count, buffer, numBytesReadPtr)
XOP_FILE_REF fileRef;           // Reference returned by XOPOpenFile
SInt64 count;                   // Count of bytes to read
void* buffer;                   // Where bytes are stored
SInt64* numBytesReadPtr;        // Output: number of bytes read
```

Reads count bytes from the referenced file into the buffer.

Use XOPReadFile64 when you need to read potentially greater than 4GB in one read call. When called from a 32-bit XOP, XOPReadFile64 is limited to 4GB and returns an error if count is greater than 4GB.

If numBytesReadPtr is not NULL, XOPReadFile64 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.

Thread Safety

XOPReadFile64 is thread-safe. It can be called from any thread.

XOPReadLine

```
int
XOPReadLine(fileRef, buffer, bufferLength, numBytesReadPtr)
XOP_FILE_REF fileRef;           // Reference returned by XOPOpenFile
char* buffer;                   // Where bytes are stored
UInt32 bufferLength;           // Size of the buffer in bytes
UInt32* 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, XOPReadLine 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.

XOPReadLine 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.

Thread Safety

XOPReadLine is thread-safe. It can be called from any thread.

XOPWriteFile

```
int
XOPWriteFile(fileRef, count, buffer, numBytesWrittenPtr)
XOP_FILE_REF fileRef;           // Reference returned by XOPOpenFile
UInt32 count;                   // Count of bytes to write
const void* buffer;             // Pointer to data to write
UInt32* numBytesWrittenPtr;     // Output: number of bytes written
```

Writes count bytes from the buffer to the referenced file.

If numBytesWrittenPtr is not NULL, XOPWriteFile stores the number of bytes written in *numBytesWrittenPtr.

The function result is 0 if OK or an error code.

Thread Safety

XOPWriteFile is thread-safe. It can be called from any thread.

XOPWriteFile64

```
int
XOPWriteFile64(fileRef, count, buffer, numBytesWrittenPtr)
XOP_FILE_REF fileRef;           // Reference returned by XOPOpenFile
SInt64 count;                   // Count of bytes to write
const void* buffer;             // Pointer to data to write
SInt64* numBytesWrittenPtr;     // Output: number of bytes written
```

Writes count bytes from the buffer to the referenced file.

Use XOPWriteFile64 when you need to write potentially greater than 4GB in one write call. When called from a 32-bit XOP, XOPWriteFile64 is limited to 4GB and returns an error if count is greater than 4GB.

If numBytesWrittenPtr is not NULL, XOPWriteFile64 stores the number of bytes written in *numBytesWrittenPtr.

The function result is 0 if OK or an error code.

Thread Safety

XOPWriteFile64 is thread-safe. It can be called from any thread.

XOPGetFilePosition

```
int
XOPGetFilePosition(fileRef, filePosPtr)
XOP_FILE_REF fileRef;           // Reference returned by XOPOpenFile
UInt32* filePosPtr;
```

Returns via filePosPtr the current file position of the referenced file.

The function result is 0 if OK or an error code.

If the file size may exceed 2 GB, use **XOPGetFilePosition2** instead of XOPGetFilePosition.

Thread Safety

XOPGetFilePosition is thread-safe. It can be called from any thread.

XOPGetFilePosition2

```
int
XOPGetFilePosition2(fileRef, dFilePosPtr)
XOP_FILE_REF fileRef;      // Reference returned by XOPOpenFile
SInt64* filePosPtr;
```

Returns via filePosPtr the current file position of the referenced file.

XOPGetFilePosition2 is the same as XOPGetFilePosition except that the file position parameter is an SInt64* rather than a UInt32* and it works with files larger than 2 GB.

The function result is 0 if OK or an error code.

Thread Safety

XOPGetFilePosition2 is thread-safe. It can be called from any thread.

XOPSetFilePosition

```
int
XOPSetFilePosition(fileRef, filePos, mode)
XOP_FILE_REF fileRef;      // Reference returned by XOPOpenFile
SInt32 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.

If the file size may exceed 2 GB, use **XOPSetFilePosition2** instead of XOPSetFilePosition.

Thread Safety

XOPSetFilePosition is thread-safe. It can be called from any thread.

XOPSetFilePosition2

```
int
XOPSetFilePosition2(fileRef, dFilePos)
XOP_FILE_REF fileRef;      // Reference returned by XOPOpenFile
SInt64 filePos;
```

Sets the current file position in the referenced file.

filePos is relative to the start of the file.

XOPSetFilePosition2 is the same as XOPSetFilePosition except that the file position parameter is an SInt64 rather than an SInt32 and it works with files larger than 2 GB and also it is lacking the mode parameter.

The function result is 0 if OK or an error code.

Thread Safety

XOPSetFilePosition2 is thread-safe. It can be called from any thread.

XOPAtEndOfFile

```
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.

Thread Safety

XOPAtEndOfFile is thread-safe. It can be called from any thread.

XOPNumberOfBytesInFile

```
int
XOPNumberOfBytesInFile(fileRef, numBytesPtr)
XOP_FILE_REF fileRef;      // Reference returned by XOPOpenFile
UInt32* numBytesPtr;
```

Returns via numBytesPtr the total number of bytes in the referenced file.

The function result is 0 if OK or an error code.

If the file size may exceed 4 GB, use **XOPNumberOfBytesInFile2** instead of XOPNumberOfBytesInFile.

Thread Safety

XOPNumberOfBytesInFile is thread-safe. It can be called from any thread.

XOPNumberOfBytesInFile2

```
int
XOPNumberOfBytesInFile2(fileRef, dNumBytesPtr)
XOP_FILE_REF fileRef;      // Reference returned by XOPOpenFile
SInt64* numBytesPtr;
```

Returns via numBytesPtr the total number of bytes in the referenced file.

XOPNumberOfBytesInFile2 is the same as XOPNumberOfBytesInFile except that the numBytesPtr parameter is an SInt64* rather than a UInt32* and it works with files larger than 4 GB.

The function result is 0 if OK or an error code.

Thread Safety

XOPNumberOfBytesInFile2 is thread-safe. It can be called from any thread.

XOPOpenFileDialog

```
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.

XOPOpenFileDialog 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.

XOPOpenFileDialog 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).

prompt sets the dialog window title.

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.

fileFilterStr on Macintosh

fileFilterStr provides control over the Enable popup menu which the Macintosh Navigation Manager displays in the Open File dialog. For example, the string:

```
"Text Files:TEXT,IGTX:.txt,.itx;All Files:****;"
```

results in two items in the Enable popup menu.

The two section sections of this fileFilterString are:

```
"Text Files:TEXT,IGTX:.txt,.itx;"
"All Files:****;"
```

Each section causes the creation of one item in the Enable popup menu.

Each section consists of three components: a menu item string to be displayed in the Enable popup menu, a list of zero or more Macintosh file types (e.g., TEXT,IGTX), and a list of extensions (e.g., .txt,.itx).

In this example, the first menu item would be "Text Files". When the user selects this menu item, the Open File dialog would show any file whose Macintosh file type is TEXT or IGTX and any file whose extension is .txt, or .itx.

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).

On Windows you can specify multiple extensions for a single type of file by listing the extensions with a semicolon between them. For example:

```
"Excel Files (*.xls,*.xlsx)\0*.xls;*.xlsx\0All Files (*.*)\0*.*\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 or Save File dialogs. 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. It is unchanged if the user cancels. 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.

Thread Safety

`XOPOpenFileDialog` is not thread-safe.

XOPSaveFileDialog

```
int
XOPSaveFileDialog(prompt, fileFilterStr, fileIndexPtr, initDir,
                  defaultExtensionStr, fullFilePath)

const char* prompt;           // Message displayed in dialog
const char* fileFilterStr;    // Controls types of files shown
int* fileIndexPtr;           // Controls initial type of file shown
const char* initDir;          // Sets initial directory
const char* defaultExtensionStr; // Default file extension or NULL
char fullFilePath[MAX_PATH_LEN+1]; // Output path returned here
```

Displays the save file dialog.

`XOPSaveFileDialog` 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.

`XOPSaveFileDialog` 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).

`prompt` sets the dialog window title.

fileFilterStr on Macintosh

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
```

`fileFilterStr` consists of sections terminated by a semicolon. For example, here is one section:

```
"Data Files:TEXT:.dat;"
```

Each section consists of three components: a menu item string (e.g., Data Files) to be displayed in the Format popup menu, a Macintosh file type (e.g., TEXT), and an extension (e.g., .dat).

At present, only the menu item string and extension are used.

The Macintosh file type is currently not used. If there is no meaningful Macintosh file type, leave the file type component empty.

If there is no meaningful extension, leave the extension component empty.

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 Enable 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 and you will get no Format menu.

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` 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 File 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` 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`. Prior to XOP Toolkit 6, `defaultExtensionStr` was ignored on Macintosh.

`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.

Thread Safety

`XOPSaveFileDialog` is not thread-safe.

Routines for File-Loader XOPs

These routines are specialized for file-loader XOPs — XOPs that load data from files into Igor waves.

GetFullPathFromSymbolicPathAndFilePath

```
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 HFS 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. It must be big enough to hold MAX_PATH_LEN+1 bytes.

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 three most common cases are:

```
LoadWave <full path to file>
LoadWave/P=<symbolic path name> <partial path to file>
LoadWave/P=<symbolic path name> <file name>
```

where <file name> connotes a simple file name.

In this case GetFullPathFromSymbolicPathAndFilePath ignores the symbolic path:

```
LoadWave/P=<symbolic path name> <full path to file>
```

In the following cases, the full path to the file can not be determined, so GetFullPathFromSymbolicPathAndFilePath 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.

Thread Safety

GetFullPathFromSymbolicPathAndFilePath is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

FileLoaderMakeWave

```
int
FileLoaderMakeWave(column, waveName, numPoints, fileLoaderFlags, whp)
int column;           // Number of column for this wave
char* waveName;       // Name for this wave
CountInt numPoints;   // Number of points in wave
int fileLoaderFlags;   // Standard file-loader flags
waveHndl* waveHPtr;   // 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.

It returns a handle to the wave via waveHPtr 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 big enough to hold MAX_OBJ_NAME+1 bytes.

Prior to XOP Toolkit 6, if the numPoints parameter was zero, the wave was created with 128 points. This behavior was a relic of a time when Igor required that a wave contain at least two points (prior to Igor Pro 3.0). Now when numPoints is zero, a zero-point wave is created.

If you find that you need more flexibility, use **MakeWave**, **MDMakeWave** or **GetOperationDestWave** instead of FileLoaderMakeWave.

Thread Safety

FileLoaderMakeWave is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetFileLoaderOutputVariables

```
int
SetFileLoaderOutputVariables(fileNameOrPath, numWavesLoaded, waveNames)
char* fileNameOrPath; // Name of or full path to 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.

Call `SetFileLoaderOutputVariables` at the end of a file load operation to set the standard file loader output globals:

Variable	Contents
<code>S_fileName</code>	The name of the file loaded
<code>S_path</code>	The full path to the folder containing the file (see description below)
<code>V_flag</code>	The number of waves loaded
<code>S_waveNames</code>	Semicolon-separate list of wave names (e.g. "wave0;wave1;")

`fileNameOrPath` 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"). Passing a full path is recommended. If it is a full path, it can be a Macintosh HFS path (using colons) or a Windows path (using backslashes).

If `fileNameOrPath` 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 HFS path syntax and includes a trailing colon.

If `fileNameOrPath` is a simple file name, `SetFileLoaderOutputVariables` does not set or create `S_path` and stores the simple file name in `S_fileName`. You should pass a full path so that `S_path` will be set.

`SetFileLoaderOutputVariables` returns 0 or an error code.

Thread Safety

`SetFileLoaderOutputVariables` is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SetFileLoaderOperationOutputVariables

int

SetFileLoaderOperationOutputVariables(**runningInUserFunction**, **fileNameOrPath**,
numWavesLoaded, **waveNames**)

```
int runningInUserFunction; // Truth that operation was called from function
const char* fileNameOrPath; // Name of or full path to the file just loaded
int numWavesLoaded;        // Number of waves created
const char* waveNames;     // Semicolon-separated list of wave names
```

Call `SetFileLoaderOperationOutputVariables` at the end of an Operation Handler file load operation to set the standard file loader output globals.

When called from the command line it creates global variables.

When called from a user-defined function or from a macro it creates local variables.

You obtain the value to pass for the `runningInUserFunction` parameter from the `calledFromFunction` field of your operation's runtime parameter structure.

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 `SimpleLoad-WaveOperation.cpp` for an example.

`SetFileLoaderOperationOutputVariables` sets the following standard file loader output globals:

Variable	Contents
<code>S_fileName</code>	The name of the file loaded.
<code>S_path</code>	The full path to the folder containing the file. See description below.

Variable	Contents
V_flag	The number of waves loaded.
S_waveNames	Semicolon-separate list of wave names (e.g. "wave0;wave1;").

fileNameOrPath 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"). Passing a full path is recommended. If it is a full path, it can be a Macintosh HFS path (using colons) or a Windows path (using backslashes).

If fileNameOrPath is a full path, SetFileLoaderOperationOutputVariables 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 HFS path syntax and includes a trailing colon.

If fileNameOrPath is a simple file name, SetFileLoaderOperationOutputVariables does not set S_path and stores the simple file name in S_fileName. You should pass a full path so that S_path will be set.

SetFileLoaderOperationOutputVariables returns 0 or error code.

Thread Safety

SetFileLoaderOperationOutputVariables is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

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.

GetXOPWindow

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.

Thread Safety

GetXOPWindow is not thread-safe.

GetActiveWindowRef

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.

Thread Safety

GetActiveWindowRef is not thread-safe.

IsXOPWindowActive

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.

Thread Safety

IsXOPWindowActive is not thread-safe.

ShowXOPWindow

void

ShowXOPWindow(windowRef)

XOP_WINDOW_REF windowRef;

Shows the specified window if it is hidden without activating it. Call this in response to the XOP_SHOW_WINDOW message from Igor.

An XOP_WINDOW_REF is a WindowPtr on Macintosh and an HWND on Windows.

Thread Safety

ShowXOPWindow is not thread-safe.

HideXOPWindow

```
void
HideXOPWindow(windowRef)
XOP_WINDOW_REF windowRef;
```

Hides the window without sending it to the bottom of the desktop. Call this in response to the XOP_HIDE_WINDOW message from Igor.

An XOP_WINDOW_REF is a WindowPtr on Macintosh and an HWND on Windows.

Thread Safety

HideXOPWindow is not thread-safe.

ShowAndActivateXOPWindow

```
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.

Thread Safety

ShowAndActivateXOPWindow is not thread-safe.

HideAndDeactivateXOPWindow

```
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.

Thread Safety

HideAndDeactivateXOPWindow is not thread-safe.

TellIgorWindowStatus

```
int
TellIgorWindowStatus(windowRef, status, options)
XOP_WINDOW_REF windowRef;
int status;
int options;
```

Call TellIgorWindowStatus when you hide or show your window, when it is activated or deactivated and when it is about to be killed.

TellIgorWindowStatus allows your window to be included various Igor features such as the Recent Windows menu item and the Show Recently Hidden Windows menu item.

It is not required that you call this routine but is recommended. If you do not call this routine then your windows will not participate in these features.

Here are the calls to make at the appropriate times:

```
TellIgorWindowStatus(windowRef, WINDOW_STATUS_DID_HIDE, 0);
TellIgorWindowStatus(windowRef, WINDOW_STATUS_DID_SHOW, 0);
TellIgorWindowStatus(windowRef, WINDOW_STATUS_ACTIVATED, 0);
TellIgorWindowStatus(windowRef, WINDOW_STATUS_DEACTIVATED, 0);
TellIgorWindowStatus(windowRef, WINDOW_STATUS_ABOUT_TO_KILL, 0);
TellIgorWindowStatus(windowRef, WINDOW_STATUS_DID_KILL, 0);
```

NOTE: If you call this routine with any status, you must also call it when your window is about to be killed so Igor can remove it from various internal lists. Failure to do so may cause a crash.

WINDOW_STATUS_DID_KILL was added in Igor Pro 6.04 but you can pass that value to any version.

See also **Activating, Showing and Hiding Windows** on page 164.

Returns 0 if OK or IGOR_OBSOLETE.

Added for Igor Pro 6.0. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

An XOP_WINDOW_REF is a WindowPtr on Macintosh and an HWND on Windows.

Thread Safety

TellIgorWindowStatus is not thread-safe.

SetXOPWindowTitle

```
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.

Thread Safety

SetXOPWindowTitle is not thread-safe.

GetXOPWindowPositionAndState

```
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.

Thread Safety

GetXOPWindowIgorPositionAndState is not thread-safe.

SetXOPWindowPositionAndState

```
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.

Thread Safety

SetXOPWindowIgorPositionAndState is not thread-safe.

TransformWindowCoordinates

```
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 162.

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.

Thread Safety

SetXOPWindowIgorPositionAndState is not thread-safe.

GetXOPWindowIgorPositionAndState

```
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 162 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 minimized 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.

Thread Safety

GetXOPWindowIgorPositionAndState is not thread-safe.

SetXOPWindowIgorPositionAndState

```
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, using coordinates stored in a preference file.

See **Igor Window Coordinates** on page 162 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 minimized 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.

Thread Safety

SetXOPWindowIgorPositionAndState is not thread-safe.

ArrowCursor

void

ArrowCursor (void)

Sets the cursor to the arrow. Your XOP can call this if it owns the top window.

Thread Safety

ArrowCursor is not thread-safe.

IBeamCursor

void

IBeamCursor (void)

Sets the cursor to the I beam. Your XOP can call this if it owns the top window.

Thread Safety

IBeamCursor is not thread-safe.

HandCursor

void

HandCursor (void)

Sets the cursor to the hand. Your XOP can call this if it owns the top window.

Thread Safety

HandCursor is not thread-safe.

WatchCursor

void

WatchCursor (void)

Sets the cursor to the watch. Your XOP can call this if it is doing a time-consuming operation.

Thread Safety

WatchCursor is not thread-safe.

SpinCursor

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**.

Thread Safety

SpinCursor is not thread-safe.

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 under **TU ("Text Utility") Windows** on page 156.

Creating and Disposing Text Windows

TUNew2

```
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.

Thread Safety

TUNew2 is not thread-safe.

TUDispose

```
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.

Thread Safety

TUDispose is not thread-safe.

Responding to Text Window Messages

TUDisplaySelection

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.

Thread Safety

TUDisplaySelection is not thread-safe.

TUGrow

void

TUGrow(TU, size)

TUStuffHandle TU; // Handle to information about the text area

int 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.

Thread Safety

TUGrow is not thread-safe.

TUUpdate

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.

Thread Safety

TUUpdate is not thread-safe.

TUActivate

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.

Thread Safety

TUActivate is not thread-safe.

TUIIdle

```
void
TUIIdle(TU)
TStuffHandle 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.

Thread Safety

TUIIdle is not thread-safe.

TUMoveToPreferredPosition

```
void
TUMoveToPreferredPosition(TU)
TStuffHandle 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.

Thread Safety

TUMoveToPreferredPosition is not thread-safe.

TUMoveToFullSizePosition

```
void
TUMoveToFullSizePosition(TU)
TStuffHandle 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.

Thread Safety

TUMoveToFullSizePosition is not thread-safe.

TURetrieveWindow

```
void
TURetrieveWindow(TU)
TStuffHandle 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.

Thread Safety

`TURetrieveWindow` is not thread-safe.

TUNull

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.

Thread Safety

`TUNull` is not thread-safe.

TUCopy

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.

Thread Safety

`TUCopy` is not thread-safe.

TUCut

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.

Thread Safety

`TUCut` is not thread-safe.

TUPaste

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.

Thread Safety

`TUPaste` is not thread-safe.

TUClear

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.

Thread Safety

TUClear is not thread-safe.

TUUndo

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.

Thread Safety

TUUndo is not thread-safe.

TUPrint

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.

Thread Safety

TUPrint is not thread-safe.

TUPageSetupDialog

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.

Thread Safety

TUPageSetupDialog is not thread-safe.

TUClick

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.

Thread Safety

TUClick is not thread-safe.

TUKey

```
void
TUKey(TU, eventPtr)
TStuffHandle 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.

Thread Safety

TUKey is not thread-safe.

TUSelectAll

```
void
TUSelectAll(TU)
TStuffHandle 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.

Thread Safety

TUSelectAll is not thread-safe.

TUFind

```
void
TUFind(TU, code)
TStuffHandle 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.

Thread Safety

TUFind is not thread-safe.

TUReplace

```
void
TUReplace(TU)
TStuffHandle 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.

Thread Safety

TUReplace is not thread-safe.

TUIndentLeft

```
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.

Thread Safety

TUIndentLeft is not thread-safe.

TUIndentRight

```
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.

Thread Safety

TUIndentRight is not thread-safe.

TUFixEditMenu

```
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.

Thread Safety

TUFixEditMenu is not thread-safe.

TUFixFileMenu

```
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.

Thread Safety

TUFixFileMenu is not thread-safe.

TUSFInsertFile

```
int
TUSFInsertFile(TU, prompt, fileType, numTypes)
TUSTuffHandle TU;           // Handle to information about the text area
char* prompt;               // C string for prompt in open dialog
OSType fileType[];          // Array of file types that you can open
short numTypes;             // Number of file types in fileType[]
```

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.

Thread Safety

TUSFInsertFile is not thread-safe.

TUSFWriteFile

```
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.

Thread Safety

TUSFWriteFile is not thread-safe.

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.

TUDrawWindow

```
void
TUDrawWindow(TU)
TUSTuffHandle TU;           // Handle to information about the text area
```

Draws the text and scroll bars in the text area.

Thread Safety

TUDrawWindow is not thread-safe.

TULines

```
int
TULines(TU)
TUSTuffHandle TU;           // Handle to information about the text area
```

Returns the total number of lines of text in the text area.

Thread Safety

TULines is not thread-safe.

TUGetDocInfo

```
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;
int paragraphs;
int result;

di.version = TUDOCINFO_VERSION;
if (result = TUGetDocInfo(theTU, &di))
    return result;
paragraphs = di.paragraphs;
```

Thread Safety

TUGetDocInfo is not thread-safe.

TUGetSelLocs

```
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.

Thread Safety

TUGetSelLocs is not thread-safe.

TUSetSelLocs

```
int
TUSetSelLocs(TU, startLocPtr, endLocPtr, flags)
TStuffHandle 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.

Thread Safety

TUSetSelLocs is not thread-safe.

TUInsert

```
void
TUInsert(TU, dataPtr, dataLen)
TStuffHandle TU;           // Handle to information about the text area
char* dataPtr;             // Pointer to text to insert
int dataLen;               // Number of characters to insert
```

Inserts the specified text in the text area.

This allows your XOP to insert text at any time.

Thread Safety

TUInsert is not thread-safe.

TUDelete

```
void
TUDelete(TU)
TStuffHandle 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.

Thread Safety

TUDelete is not thread-safe.

TUFetchParagraphText

```
int
TUFetchParagraphText(TU, paragraph, textPtrPtr, lengthPtr)
TUStuffHandle TU;           // Handle to information about the text area
int paragraph;              // Number of paragraph to fetch starting from 0
char* textPtrPtr;           // Receives pointer to text
int* 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 using **DisposePtr**.

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;
int paragraph, length;
int result;

paragraph = 0;
if (result = TUFetchParagraphText(TU, paragraph, &p, &length))
    return result;
<Deal with the text pointed to by p>
DisposePtr((Ptr)p);
```

Note that the text pointed to by p is not null terminated and therefore is not a C string.

Thread Safety

TUFetchParagraphText is not thread-safe.

TUFetchSelectedText

```
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
int 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 237.

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;

if (result = TUFetchSelectedText(TU, &h, NULL, 0))
    return result;
<Deal with the text in handle>
DisposeHandle(h);

```

Thread Safety

TUFetchSelectedText is not thread-safe.

TUFetchText2

```

int
TUFetchText2(TU, startLocPtr, endLocPtr, textHandlePtr, reservedForFuture, flags)
TUSuffHandle TU;           // Handle to information about the text area
TULocPtr startLocPtr;      // Start of text to fetch
TULocPtr endLocPtr;        // End of text to fetch
Handle* textHandlePtr;     // Receives handle containing text
void* reservedForFuture;   // Pass NULL for this
int flags;                 // Miscellaneous flags

```

Returns via textHandlePtr the text utility document from the start location to the end location.

If startLocPtr is NULL, the start of the document is used as the start location.

If endLocPtr is NULL, the end of the document is used as the end location.

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 237.

reservedForFuture should be NULL for now.

flags should be 0 for now.

TUFetchText2 was added in Igor Pro 6.20 but works with any version.

The function result is 0 if OK or an error code if an error occurs fetching the text.

Example

```

Handle h;
int result;

// Fetch all text in document
if (result = TUFetchText2(TU, NULL, NULL, &h, NULL, 0))
    return result;
<Deal with the text in handle>
DisposeHandle(h);

```

Thread Safety

TUFetchText2 is not thread-safe.

TUSetStatusArea

```
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.

Thread Safety

TUSetStatusArea is not thread-safe.

Routines for Dealing with Resources

These callbacks are useful for accessing resources from within an XOP.

XOPRefNum

```
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.

Thread Safety

XOPRefNum is thread-safe but there is little if anything you can do with the returned reference number that is thread-safe.

GetXOPResource

```
Handle
GetXOPResource(resType, resID)
int 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.

Thread Safety

GetXOPResource is not thread-safe.

GetXOPNamedResource

```
Handle
GetXOPNamedResource(resType, name)
int 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.

Thread Safety

GetXOPNamedResource is not thread-safe.

GetXOPIndString

```
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.

Thread Safety

GetXOPIndString is not thread-safe.

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 superseded 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.

GetNamedFIFO

```
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.

Thread Safety

GetNamedFIFO is not thread-safe.

MarkFIFOUpdated

```
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.

Thread Safety

MarkFIFOUpdated is not thread-safe.

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
SInt32 (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
UInt32 (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.

ConvertData

```
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
CountInt 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, SInt32, Short, Byte, UInt32, UnsignedShort, UnsignedByte

For most uses, you should use the higher level ConvertData routine.

Thread Safety

ConvertData is thread-safe. It can be called from any thread.

ConvertData2

```

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
CountInt 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).

Thread Safety

ConvertData2 is thread-safe. It can be called from any thread.

NumTypeToNumBytesAndFormat

```
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.

Thread Safety

NumTypeToNumBytesAndFormat is thread-safe. It can be called from any thread.

NumBytesAndFormatToNumType

```
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.

Thread Safety

NumBytesAndFormatToNumType is thread-safe. It can be called from any thread.

ScaleData

```
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
CountInt 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, SInt32, Short, Byte, UInt32, UnsignedShort, UnsignedByte

For most uses, you should use the high level ScaleData routine.

Thread Safety

ScaleData is thread-safe. It can be called from any thread.

ScaleClipAndRoundData

```
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
CountInt 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.

Thread Safety

ScaleClipAndRoundData is thread-safe. It can be called from any thread.

FixByteOrder

```
void
FixByteOrder(p, bytesPerPoint, numValues)
void* p;                // Pointer to input data of any type
int bytesPerPoint;      // Bytes in one point of input format
CountInt numValues;     // Number of values to fix
```

The routine converts data from Motorola (high byte first) to Intel (low byte first) format or vice-versa.

Thread Safety

FixByteOrder is thread-safe. It can be called from any thread.

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 **CreateValidDataObjectName** 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 **XOPCommand**, **XOPCommand2**, **XOPCommand3**, **XOPSilentCommand** or **FinishDialogCmd** you must quote liberal object names. Use **PossiblyQuoteName** and **CatPossiblyQuotedName** for this purpose. See the Igor Pro manual for general information on liberal names.

UniqueName

```
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.

Thread Safety

UniqueName is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

UniqueName2

```
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
int 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.

Thread Safety

UniqueName2 is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

SanitizeWaveName

```
int
SanitizeWaveName(waveName, column)
char* waveName;           // Input and output wave name
int 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.

Thread Safety

SanitizeWaveName is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

CleanupName

```
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.

Wave and data folder names can contain characters, such as space and dot. 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.

Thread Safety

CleanupName is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

CheckName

```
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.

Thread Safety

CheckName is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

CreateValidDataObjectName

```
int
CreateValidDataObjectName(
DataFolderHandle dataFolderH, // Handle to data folder or NULL
char* inName,                // Input: Proposed name
char* outName,                // Output: Valid name
int* 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 `CreateValidDataObjectName`.

`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.

```
IndexInt column;
int 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;
}
}

```

Thread Safety

CreateValidDataObjectName is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

PossiblyQuoteName

```

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.

A wave or data folder name can be either a "standard" name or a "liberal" name. Standard names start with a letter and contain letters, digits and underscore characters only. Liberal names can contain spaces and punctuation.

When a liberal name is used in Igor's command line it must be enclosed in single quotes. This includes using it in the Execute operation or in the **XOPCommand**, **XOPCommand2**, **XOPCommand3**, **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.

Thread Safety

PossiblyQuoteName is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

CatPossiblyQuotedName

```

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);
```

Thread Safety

CatPossiblyQuotedName is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

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 Gizmo 3D plotting XOP which allows the user to choose from a set of Igor color tables.

The Gizmo XOP 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 **GetNamedIgorColorTableHandle** to get a color table handle, **GetIgorColorTableInfo** to find the number of colors in the table, and **GetIgorColorTableValues** to find the actual colors.

GetIndexedIgorColorTableName

```
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.

Thread Safety

GetIndexedIgorColorTableName is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetNamedIgorColorTableHandle

```
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, Red-WhiteBlue, 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.

Thread Safety

GetNamedIgorColorTableHandle is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetIgorColorTableInfo

```
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.

Thread Safety

GetIgorColorTableInfo is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

GetIgorColorTableValues

```
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.

Thread Safety

GetIgorColorTableValues is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

Command Window

These routines support interacting with the command window.

HistoryDisplaySelection

void

HistoryDisplaySelection(void)

Scrolls the current selection in the history area into view.

Added for Igor Pro 6.00B10. Does nothing in earlier versions of Igor.

Thread Safety

HistoryDisplaySelection is not thread-safe.

HistoryInsert

void

HistoryInsert(dataPtr, dataLen)

char* dataPtr; // Pointer to text to insert

int dataLen; // Number of characters to insert

Inserts the specified text into the history area, replacing the current selection, if any.

If you just want to append text to the history, call **XOPNotice**, **XOPNotice2** or **XOPNotice3** instead of **HistoryInsert**.

Except in very rare cases you should not modify the history, except to append to it.

Added for Igor Pro 6.00B10. Does nothing in earlier versions of Igor.

Thread Safety

HistoryInsert is not thread-safe.

HistoryDelete

void

HistoryDelete(void)

Deletes the currently selected text in the history area.

Except in very rare cases you should not modify the history, except to append to it.

Added for Igor Pro 6.00B10. Does nothing in earlier versions of Igor.

Thread Safety

HistoryDelete is not thread-safe.

HistoryLines

int

HistoryLines(void)

Returns the total number of lines of text in the history area.

Added for Igor Pro 6.00B10. Returns -1 in pre-Igor Pro 6 versions of Igor.

Thread Safety

HistoryLines is not thread-safe.

HistoryGetSelLocs

```
int
HistoryGetSelLocs(startLocPtr, endLocPtr)
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 history area.

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.

Returns 0 if OK or an Igor error code.

Added for Igor Pro 6.00B10. Returns IGOR_OBSOLETE in earlier versions of Igor.

Thread Safety

HistoryGetSelLocs is not thread-safe.

HistorySetSelLocs

```
int
HistorySetSelLocs(startLocPtr, endLocPtr, flags)
TULocPtr startLocPtr;    // Contains location of start of selection
TULocPtr endLocPtr;      // Contains location of end of selection
int flags;               // Miscellaneous flags
```

Sets the selection in the history area.

If startLocPtr is NULL, the start location is taken to be the start of history area.

If endLocPtr is NULL, the end location is taken to be the end of history area.

If flags is 1, displays the selection if it is out of view.

Other bits in flags must be set to 0 as they may be used for other purposes in the future.

Returns 0 if OK or an Igor error code.

Returns an error if the start or end locations are out of bounds or if the start location is after the end location.

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 history area as reported by HistoryLines. The character position field must be between 0 and c where c is the number of characters in the paragraph as reported by **HistoryFetchParagraphText**. 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.

Added for Igor Pro 6.00B10. Returns IGOR_OBSOLETE in earlier versions of Igor.

Thread Safety

HistorySetSelLocs is not thread-safe.

HistoryFetchParagraphText

```
int
HistoryFetchParagraphText(paragraph, textPtrPtr, lengthPtr)
int paragraph;           // Number of paragraph to fetch starting from 0
char* textPtrPtr;        // Receives pointer to text
int* 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. If it is not NULL, Igor allocates a pointer, using **NewPtr**, and sets *textPtrPtr to point to the allocated memory. You must dispose this when you no longer need it using **DisposePtr**.

Note that the text returned via textPtrPtr is *not* null terminated.

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.

Added for Igor Pro 6.00B10. Returns IGOR_OBSOLETE in earlier versions of Igor.

Example

```
char* p;
int paragraph, length;
int result;

paragraph = 0;
if (result = HistoryFetchParagraphText(paragraph, &p, &length))
    return result;
<Deal with the text pointed to by p>
DisposePtr((Ptr)p);
```

Thread Safety

HistoryFetchParagraphText is not thread-safe.

HistoryFetchText

```
int
HistoryFetchText(startLocPtr, endLocPtr, textHPtr)
TULocPtr startLocPtr;      // Identifies start of text to fetch
TULocPtr endLocPtr;        // Identifies end of text to fetch
Handle* textHPtr;          // Receives handle containing text
```

Returns the history area text from the specified start location to the specified end location.

If startLocPtr is NULL, the start location is taken to be the start of history area.

If endLocPtr is NULL, the end location is taken to be the end of history area.

On return, if there is an error, *textHPtr will be NULL. If there is no error, *textHPtr will point to a handle containing the text. *textHPtr belongs to you so dispose it when you are finished with it using **DisposeHandle**.

Note that the text in the handle is *not* null terminated. See **Understand the Difference Between a String in a Handle and a C String** on page 237.

The function result is 0 if OK, an error code if an error occurs fetching the text.

Added for Igor Pro 6.00B10. Returns IGOR_OBSOLETE in earlier versions of Igor.

Example

```
Handle h;
int result;

if (result = HistoryFetchSelectedText(NULL, NULL, &h))
    return result;
<Deal with the text in handle>
DisposeHandle(h);
```

Thread Safety

HistoryFetchSelectedText is not thread-safe.

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.

GetIgorProcedureList

```
int
GetIgorProcedureList(Handle* hPtr, flags)
Handle* hPtr;
int 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 237.

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	Meaning
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.

Thread Safety

GetIgorProcedureList is not thread-safe.

GetIgorProcedure

```
int
GetIgorProcedure(procedureName, hPtr, flags)
const char* procedureName;
Handle* hPtr;
int 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`, `GetIgorProcedure` 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 237. 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	Meaning
Bit 0	If set, <code>GetIgorProcedure</code> will look for macros with the specified name. If cleared, it will ignore all macros.
Bit 1	If set, <code>GetIgorProcedure</code> 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.

Thread Safety

`GetIgorProcedure` is not thread-safe.

SetIgorProcedure

```
int
SetIgorProcedure(procedureName, h, flags)
const char* procedureName;
Handle h;
int flags;
```

This routine is used by very advanced XOPs, like the Gizmo 3D 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 164 for further information.

Thread Safety

`SetIgorProcedure` is not thread-safe.

DoWindowRecreationDialog

```
enum CloseWinAction
```

```
DoWindowRecreationDialog(procedureName)
```

```
char* procedureName;
```

This routine is used by very advanced XOPs, like the Gizmo 3D plotter, that add a target window type to Igor. See **Adding XOP Target Windows** on page 164 for further information.

Thread Safety

DoWindowRecreationDialog is not thread-safe.

GetFunctionInfo

```
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 213.

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.

When you call GetFunctionInfo with a simple function name, such as "MyFunction", the information returned describes the named function in the currently-executing independent module. If you call GetFunctionInfo with a fully-qualified name, such as "MyModule#MyFunction", the information returned describes the named function in the named independent module. So if you want to call from one independent module to another using **CallFunction**, you must use fully-qualified names when calling GetFunctionInfo.

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
WAVE_TYPE:	Return value is a wave reference
DATAFOLDER_TYPE:	Return value is a DFREF

Wave reference and DFREF return types are supported in Igor Pro 6.10 or later.

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.

Thread Safety

GetFunctionInfo is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

The FunctionInfo structure contains an `isThreadSafe` field which tells you if the function in question is threadsafe.

GetFunctionInfoFromFuncRef

```
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 213.

`fRef` is the value of a `FUNCREF` field in an Igor Pro structure passed to your XOP as a parameter.

GetFunctionInfoFromFuncRef does not support cross-independent-module calls. It assumes that the `FUNCREF` passed to it was created in the module that is currently running. There is currently no way to use a `FUNCREF` to make cross-independent-module calls. Use **GetFunctionInfo** with a qualified name instead.

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.

Thread Safety

GetFunctionInfoFromFuncRef is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

The FunctionInfo structure contains an `isThreadSafe` field which tells you if the function in question is threadsafe.

CheckFunctionForm

```
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 213.

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
DATAFOLDER_TYPE	The parameter must be a DFREF
FV_FUNC_TYPE	The parameter must be a function reference
FV_STRUCT_TYPE FV_REF_TYPE	The parameter must be a structure

DFREF parameters are supported in Igor Pro 6.10 or later.

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.

For background information see **Structure Parameters** on page 87.

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 87 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
WAVE_TYPE
DATAFOLDER_TYPE

```

Wave reference and DFREF return types are supported in Igor Pro 6.10 or later.

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.

Thread Safety

CheckFunctionForm is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

CallFunction

```

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 213.

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 209.

Parameter type values are discussed in detail in the comments for the GetFunctionInfo function in XOP-Support.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 (DATAFOLDER_TYPE bit is set) // Igor Pro 6.10 or later
    structure field is DataFolderHandle

if (FV_FUNC_TYPE bit is set)
    structure field is PSInt

```

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`:

<code>NT_FP64</code>	<code>double</code>
<code>NT_FP64 NT_CMPLX</code>	<code>double [2]</code>
<code>HSTRING_TYPE</code>	<code>Handle</code>
<code>WAVE_TYPE</code>	<code>waveHndl</code>
<code>DATAFOLDER_TYPE</code>	<code>DataFolderHandle</code>

Wave reference and `DFREF` return types are supported in Igor Pro 6.10 or later.

NOTE: A function that returns a string, wave or `DataFolderHandle` 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.

NOTE: For string return values, 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.

Thread Safety

`CallFunction` is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

You can call a thread-safe user-defined function or external function while you are running an Igor preemptive thread. You can tell if you are running in a thread by calling **RunningInMainThread**.

When you call **GetFunctionInfo**, you receive information about the function in a `FunctionInfo` structure. The `FunctionInfo` structure contains an `isThreadSafe` field. If you are running in a thread, you can check the `isThreadSafe` field to make sure the function you are calling is threadsafe.

If you try to call a non-threadsafe function from a thread using `CallFunction`, Igor will return an error, typically the `ONLY_THREADSAFE` ("Function must be ThreadSafe") error.

GetIgorCallerInfo

```
int
GetIgorCallerInfo(pathOrTitle, linePtr, routineName, callStackHPtr)
char pathOrTitle[MAX_PATH_LEN+1];
int* linePtr;
char routineName[256];
Handle* callStackHPtr;
```

Returns information about the Igor procedure that is currently running. This may be useful for debugging purposes in complex Igor projects.

If the currently running procedure is stored in a standalone file then the full path to the procedure file is returned in Macintosh HFS format (using colon separators) via pathOrTitle.

If the currently running procedure is in the built-in procedure window or in a packed procedure file or in a new procedure file not yet saved to disk then the window's title is returned via pathOrTitle preceded by a colon (e.g., ":Procedure").

If no procedure is running then pathOrTitle is set to an empty string ("").

If a procedure is running then *linePtr is set to the zero-based line number within the file.

When a macro is running this returns the line number of the macro declaration.

When a user-defined function is running it returns the line number of the statement which is currently executing.

If no procedure is running then *linePtr is set to -1.

If a procedure is running then the name of the procedure is returned via routineName. Otherwise routineName is set an empty string ("").

If GetIgorCallerInfo returns zero then the function has succeeded and you own the handle referenced by callStackHPtr, and you must dispose it.

If GetIgorCallerInfo returns a non-zero error code then *stackInfoHPtr is undefined and you must not do anything with it.

If a procedure is running, *callStackHPtr is set to a handle containing a stack crawl.

If no procedure is running, *callStackHPtr is set to NULL. You should always test *callStackHPtr - do not use it if it is NULL. If *callStackHPtr is not NULL then this handle belongs to you and you must dispose it when you no longer need it.

If *callStackHPtr is not NULL it contains a list of Igor procedures separated by semicolons. *callStackHPtr is not null terminated. Use **GetHandleSize** to determine the size of the stack crawl text.

In some cases the currently running procedure can not be determined, for example because because Igor procedures need to be compiled. If so then GetIgorCallerInfo acts as if no procedure is running.

Added for Igor Pro 6.00B09. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

Thread Safety

GetIgorCallerInfo is not thread-safe.

GetIgorRTStackInfo

```
int
GetIgorRTStackInfo(code, stackInfoHPtr)
Int code;
Handle* stackInfoHPtr;
```

Returns information about the chain of Igor procedure that are currently running. This may be useful for debugging purposes in complex Igor projects.

If GetIgorRTStackInfo returns zero then the function has succeeded and you own the handle referenced by stackInfoHPtr, and you must dispose it.

If GetIgorRTStackInfo returns a non-zero error code then *stackInfoHPtr is undefined and you must not do anything with it.

The contents of the string returned by GetIgorRTStackInfo is the same as the contents of the string returned by the built-in Igor function GetRTStackInfo. See the documentation for GetRTStackInfo for details.

The string stored in *stackInfoHPtr is not null terminated. Use **GetHandleSize** on *stackInfoHPtr to determine the size of the text.

Added for Igor Pro 6.02B01. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

Thread Safety

GetIgorRTStackInfo is not thread-safe.

Multithreading Support Routines

These routines support multithreading.

RunningInMainThread

```
int
RunningInMainThread(void)
```

Returns 1 if you are running in the main thread, 0 if you are running in another thread.

If you are not running in the main thread then certain calls are illegal. For example, an external operation or function can not do a callback to Igor if it is running with an Igor Pro version prior to 6.20. Here is code that enforces this:

```
if (igorVersion < 620) {
    if (!RunningInMainThread())
        return NOT_IN_THREADSAFE; // NOT_IN_THREADSAFE is an Igor error code.
}
```

For general information on multithreaded XOPs, see Chapter 10, **Multithreading**.

Thread Safety

RunningInMainThread is thread-safe. It can be called from any thread.

CheckRunningInMainThread

```
int
CheckRunningInMainThread(routineName)
const char* routineName; // Name of the calling routine.
```

CheckRunningInMainThread is called by non-thread-safe XOPSupport routines to provide the XOP programmer with feedback if the XOP calls a non-threadsafe XOPSupport routine from a thread other than the main thread. In this event, it displays an error dialog.

To avoid a cascade of such dialogs, the dialog is presented only once per minute.

The return value is the truth that you are running in the main thread.

You can call CheckRunningInMainThread from your own code to provide feedback if your non-thread-safe routine is called from a pre-emptive thread. It is typically used as follows:

```
if (!CheckRunningInMainThread("SomeRoutineName"))
    return NOT_IN_THREADSAFE; // NOT_IN_THREADSAFE is an Igor error code.
}
```

For general information on multithreaded XOPs, see Chapter 10, **Multithreading**.

Thread Safety

CheckRunningInMainThread is not thread-safe but can be used to alert the XOP programmer if a non-threadsafe function is mistakenly called from a preemptive thread.

ThreadGroupPutDF

```
int
ThreadGroupPutDF(threadGroupID, dataFolderH)
int threadGroupID;
DataFolderHandle dataFolderH;
```

ThreadGroupPutDF is used to pass data folders between Igor threads by way of an Igor-maintained queue. It snips the specified data folder from the data hierarchy of the calling thread and posts it to a queue from which it can be retrieved by calling **ThreadGroupGetDF** from a different thread.

When called from the main thread, ThreadGroupPutDF removes the specified data folder from the main thread's data hierarchy and posts it to the input queue of the Igor thread group specified by threadGroupID. You would do this to pass a data folder to an Igor preemptive thread for processing.

When called from an Igor preemptive thread, ThreadGroupPutDF removes the specified data folder from the running thread's data hierarchy and posts it to the output queue of the thread group. You would do this to pass data from an Igor preemptive thread to the main thread. In this case, threadGroupID is ignored and you can pass 0 for it.

threadGroupID is a thread group ID passed to you from a calling user-defined function or 0 as described above. See the Igor documentation for ThreadGroupCreate for further discussion of thread group IDs.

dataFolderH is a handle to the data folder to be removed from the currently-running Igor thread's data hierarchy and transferred to the thread group's input queue or output queue as described above.

dataFolderH must be a valid data folder handle. It must not be NULL.

If ThreadGroupPutDF succeeds, the specified data folder handle is no longer valid and must not be further used.

See the Igor documentation for the ThreadGroupPutDF operation for further details and warnings.

For general information on multithreaded XOPs, see Chapter 10, **Multithreading**.

Added for Igor Pro 6.23. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

Thread Safety

ThreadGroupPutDF is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.23 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

ThreadGroupGetDF

```
int
ThreadGroupGetDF(threadGroupID, waitMilliseconds, dataFolderHPtr)
int threadGroupID;
int waitMilliseconds;    // Milliseconds to wait for data folder
DataFolderHandle* dataFolderHPtr;
```

ThreadGroupGetDF is used to pass data folders between Igor threads by way of an Igor-maintained queue. It removes a data folder from the queue, if one is available, and stitches it into the data hierarchy of the currently running Igor thread.

When called from the main thread, ThreadGroupGetDF returns a data folder from the output queue of the thread group specified by threadGroupID or NULL if there are no data folders in the queue. If a data folder is available, it is stitched into the main thread's data hierarchy. You would do this to move results from an Igor preemptive thread into the main thread for display and further processing.

When called from an Igor preemptive thread, ThreadGroupGetDF returns a data folder from the input queue of the currently running Igor thread's thread group or NULL if there are no data folders in the queue. You would do this to get input from the main thread into the Igor preemptive thread. If a data folder

is available, it is stitched into the currently-running Igor thread's data hierarchy. In this case, thread-GroupID is ignored and you can pass 0 for it.

threadGroupID is a thread group ID passed to you from a calling user-defined function or 0 as described above. See the Igor documentation for ThreadGroupCreate for further discussion of thread group IDs.

waitMilliseconds is the number of milliseconds to wait to see if a data folder becomes available in the queue. Pass 0 to test if a data folder is available immediately. The value for waitMilliseconds must be between 0 and 2147483647 ($2^{31}-1$ milliseconds - about 25 days).

*dataFolderHPtr is set to NULL if no data folders are available or to a handle to the data folder that was retrieved from the queue. Igor changes the name of the data folder if that is necessary to make it unique within its parent data folder.

On return, check the function result from ThreadGroupGetDF and *dataFolderHPtr. If the function result is 0 and *dataFolderHPtr is not NULL then *dataFolderHPtr is a valid data folder handle in the currently-running thread's data hierarchy.

See the Igor documentation for the ThreadGroupGetDF operation for further details and warnings.

For general information on multithreaded XOPs, see Chapter 10, **Multithreading**.

Added for Igor Pro 6.23. If you call this with an earlier version of Igor, it will return IGOR_OBSOLETE and do nothing.

Thread Safety

ThreadGroupGetDF is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.23 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

Windows-Specific Routines

These routines are available and applicable to XOPs running under the Windows OS only.

XOPModule

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.

Thread Safety

XOPModule is thread-safe. It can be called from any thread.

IgorModule

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.

Thread Safety

IgorModule is thread-safe. It can be called from any thread.

IgorClientHWND

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.

Thread Safety

IgorClientHWND is thread-safe. It can be called from any thread.

WMGetLastError

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 58.

Thread Safety

WMGetLastError is thread-safe. It can be called from any thread. It relies on the Windows GetLastError function which uses per-thread storage to store an error code for each thread.

WindowsErrorToIgorError

```
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 58.

Thread Safety

WindowsErrorToIgorError is thread-safe. It can be called from any thread.

SendWinMessageToIgor

```
int
SendWinMessageToIgor(HWND hwnd, int iMsg, WPARAM wParam, LPARAM 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). After: Nothing.
WM_DESTROY	Before: Nothing. After: Deallocates memory allocated by WM_CREATE.
WM_MDIACTIVATE (when XOP window is being activated only)	Before: Compiles procedure windows if necessary. After: Sets Igor menu bar (e.g., removes "Graph" menu from Igor menu bar).

See **Adding a Simple Window on Windows** on page 154 for background information.

Thread Safety

SendWinMessageToIgor is not thread-safe.

Miscellaneous Routines

These routines don't fit in any particular category.

XOPCommand

```
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 null-terminated C string. 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** for a definition of "update".

XOPCommand displays the command in Igor's command line while it is executing. If you don't want the command to be shown in the command line, which is usually the case, use **XOPCommand2**, **XOPCommand3** or **XOPSilentCommand** instead of XOPCommand.

XOPCommand does not send the executed command to the history area.

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 64 for details.

Thread Safety

XOPCommand is not thread-safe.

XOPCommand2

```
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 null-terminated C string. 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 sendToHistory is non-zero and the command generates no error, the command is sent to the history area after execution.

A side-effect of XOPCommand2 is that it causes Igor to do an update. See **DoUpdate** below for a definition of "update".

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 64 for details.

Thread Safety

XOPCommand2 is not thread-safe.

XOPCommand3

```
int
XOPCommand3(cmdPtr, silent, sendToHistory, historyTextHPtr)
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
Handle* historyTextHPtr; // Receives history text generated by command.
```

This is just like **XOPCommand2** except that it returns via historyTextHPtr a handle containing any text added to the history by the command. In the event of an error, *historyTextHPtr is set to NULL. If no error occurs, *historyTextHPtr is set to a new handle which you own and must dispose using **DisposeHandle**. *historyTextHPtr is not null terminated.

The function result is 0 if OK or an error code.

Added in Igor Pro 6.00B10. If you call this when running with an earlier version, it will return IGOR_OBSOLETE.

Thread Safety

XOPCommand3 is not thread-safe.

XOPSilentCommand

```
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** for a definition of “update”.

This is just like **XOPCommand** except that it does not display the command in Igor’s command line as it is executing.

XOPCommand does not send the executed command to the history area.

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 64 for details.

Thread Safety

XOPSilentCommand is not thread-safe.

DoUpdate

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 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 64 for details.

Thread Safety

`DoUpdate` is not thread-safe.

PauseUpdate

void

PauseUpdate (savePtr)

int* 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.

Thread Safety

`PauseUpdate` is not thread-safe.

ResumeUpdate

void

ResumeUpdate (savePtr)

int* savePtr; // Previous state from PauseUpdate call

Undoes effect of previous `PauseUpdate`.

Make sure this is balanced with a previous `PauseUpdate` call.

Thread Safety

`ResumeUpdate` is not thread-safe.

XOPNotice

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.

`XOPNotice` prints only the first `MAXCMDLEN-2` (currently 398) bytes of the specified message.

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.

Thread Safety

XOPNotice is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

Igor queues text sent to the history area from a preemptive thread and sends the text to the history area in the main thread. Consequently text from different threads will appear in the history in unpredictable order.

XOPNotice2

```
void
XOPNotice2(noticePtr, options)
char* noticePtr;           // Message for Igor's history area
UInt32 options;
```

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.

XOPNotice2 is the same as **XOPNotice** except that it allows you to prevent marking the current experiment to be marked as modified.

options is defined as follows:

Bit	Meaning
Bit 0	If cleared, the modification state of the history area is not modified. Use this if you want to display a notice but don't want that to cause the current experiment to be modified. If set, the history area is marked as modified after the notice is displayed.
All other bits	Reserved. You must pass zero for these bits.

XOPNotice2 prints only the first MAXCMDLEN-2 (currently 398) bytes of the specified message.

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 XOPNotice2. The reason for this is that the act of doing the callback overwrites the message and arguments that Igor is passing to you.

Thread Safety

XOPNotice2 is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

Igor queues text sent to the history area from a preemptive thread and sends the text to the history area in the main thread. Consequently text from different threads will appear in the history in unpredictable order.

XOPNotice3

```
void
```

```
XOPNotice3(noticePtr, rulerName, options)
```

```
char* noticePtr;           // Message for Igor's history area
```

```
char* rulerName;           // Name of ruler in history carbon copy
```

```
int options;
```

Displays the C string pointed to by noticePtr in Igor's history.

Through the rulerName and options parameters, XOPNotice3 provides control over the formatting of the text in the History Carbon Copy notebook. If you are not using the History Carbon Copy feature (added in Igor Pro 6.10B04), use XOPNotice instead of XOPNotice3.

noticePtr is a null-terminated C string. Typically notices should be terminated by carriage return (use CR_STR at end of string as shown below).

rulerName is the name of a ruler in the History Carbon Copy notebook. XOPNotice3 applies the ruler to the current selection in the History Carbon Copy notebook before inserting the text. If rulerName is "" then the ruler is not changed.

options is defined as follows:

Bit	Meaning
Bit 0	If set, the history carbon copy ruler is restored after the text is inserted. If cleared, the history carbon copy ruler is not restored.
All other bits	Reserved. You must pass zero for these bits.

Typically you would call this to use a ruler that you previously created in the history carbon copy notebook to control formatting for specific kinds of notices. For example, if you created a ruler named Warning, you would send a warning to the history like this:

```
XOPNotice3("You have been warned!"CR_STR, "Warning", 1);
```

Bit 0 of options should be set except in the case when you expect to make many calls to XOPNotice3 in a row. In this case, you can make it run slightly faster like this:

```
XOPNotice3("Starting", "Progress", 0);           // Starting a long process
XOPNotice3(" .", "", 0);                         // Add a dot to indicate progress
. . .
XOPNotice3(" .", "", 0);                         // Add a dot to indicate progress
XOPNotice3("Done!"CR_STR, "", 0);                // The long process is done
XOPNotice3("", "Normal", 0);                     // Restore to normal ruler
```

In this example, we set the history carbon copy ruler to "Progress" which is the name of a ruler that we created for paragraphs containing progress messages. We then called XOPNotice3 repeatedly to add a dot to the progress message. In these calls, we specified no ruler so the ruler is not changed. At the end, we add a carriage return (CR_STR) to make sure we are on a new paragraph and then set the ruler to Normal so that normal history carbon copy processing will occur in the future.

XOPNotice3 prints only the first MAXCMDLEN-2 (currently 398) bytes of the specified message.

Added in Igor Pro 6.10B04. If you call this when running with an earlier version, nothing will be sent to the history.

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 XOPNotice3. The reason for this is that the act of doing the callback overwrites the message and arguments that Igor is passing to you.

Thread Safety

XOPNotice3 is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

Igor queues text sent to the history area from a preemptive thread and sends the text to the history area in the main thread. Consequently text from different threads will appear in the history in unpredictable order.

XOPResNotice

```
void
XOPResNotice(strListID, index)
int strListID;           // Resource ID to get string from
int index;               // String number in that resource
```

NOTE: XOPResNotice is antiquated. Use **XOPNotice** or **XOPNotice2** instead.

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.

XOPResNotice prints only the first MAXCMDLEN-2 (currently 398) bytes of the specified message.

Thread Safety

XOPResNotice is not thread-safe.

WaveList

```
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 237. 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.

Thread Safety

WaveList is not thread-safe.

WinList

```
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 237. 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.

Thread Safety

WinList is not thread-safe.

PathList

```
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 237. 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.

Thread Safety

PathList is not thread-safe.

GetPathInfo2

```
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.

Thread Safety

GetPathInfo2 is not thread-safe.

VariableList

```
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. 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.

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 237. 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.

Thread Safety

VariableList is not thread-safe.

StringList

```
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 237. 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.

Thread Safety

StringList is not thread-safe.

CheckAbort

```
int
CheckAbort(timeoutTicks)
TickCountInt 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 (Macintosh) or Ctrl-Break (Windows).

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.

CheckAbort *only* checks if the abort key combinations listed above are pressed. It does not check if an abort occurs in Igor which would happen if the user presses the abort keys or clicks the Abort button in the status bar on Windows or if a procedure calls the Abort operation. For most purposes, use **SpinProcess**, which checks for all types of aborts, instead of CheckAbort. Use CheckAbort if SpinProcess, which calls back to Igor, would be too slow, or if you are running in a preemptive thread with a version of Igor prior to 6.20 in which case SpinProcess is not thread-safe.

On Macintosh this routine may not work correctly on non-English keyboards. It checks the key that is the dot key on an English keyboard. This is the key that is two keys to the left of the righthand shift key. On a French keyboard, for example, that is the colon key so you must press cmd-colon instead of cmd-dot.

If this is a problem, you may prefer to use the CheckEventQueueForUserCancel function from the Carbon library instead of CheckAbort. However CheckEventQueueForUserCancel checks for the escape key in addition to cmd-dot. It also removes events from the event queue which may prevent Igor from detecting that you want to abort procedure execution.

See **Checking For Aborts** on page 216 for further discussion.

Thread Safety

CheckAbort is thread-safe. It can be called from any thread.

IgorError

```
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 57 for details.

Thread Safety

IgorError is not thread-safe.

GetIgorErrorMessage

```
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 calling 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 57 for details.

Thread Safety

GetIgorErrorMessage is not thread-safe.

SpinProcess

```
int
SpinProcess(void)
```

When running in the main thread, SpinProcess spins the beach ball cursor, updates progress windows and does other tasks that need to be done periodically.

Whether called from the main thread or from an Igor preemptive thread, SpinProcess returns the truth that your external operation or function should abort because:

The user has requested an abort

A user-defined function called the Abort operation

An error has occurred that requires an abort

To just spin the beachball cursor, 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 64 for details.

See **Checking For Aborts** on page 216 for further discussion.

Thread Safety

SpinProcess is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

PutCmdLine

```
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:

Mode	Meaning
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 FIRSTCMD-CRHIT 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**, **XOPCommand2**, **XOPCommand3** or **XOPSilentCommand** instead.

Thread Safety

PutCmdLine is not thread-safe.

XOPDisplayHelpTopic

```
int
XOPDisplayHelpTopic(title, topicStr, flags)
const char* title;           // Title for help window
const char* topicStr;        // The help topic to be displayed
int flags;
```

Displays help from an Igor help file for the specified topic. See **Igor Pro Help File** on page 219 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	Meaning
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 to 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.

Thread Safety

XOPDisplayHelpTopic is not thread-safe.

XOPSetContextualHelpMessage

```
int
XOPSetContextualHelpMessage(theWindow, msg, r)
XOP_WINDOW_REF theWindow;      // WindowRef on Mac, HWND on Windows
const char* message;           // 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.

Thread Safety

XOPSetContextualHelpMessage is not thread-safe.

WinInfo

```
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.

Thread Safety

WinInfo is not thread-safe.

SaveXOPPrefsHandle

```
int  
SaveXOPPrefsHandle(prefsHandle)  
Handle prefsHandle;          // Handle containing prefs data.
```

Saves the handle in Igor's preferences file. You can retrieve the handle using **GetXOPPrefsHandle**.

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.

Thread Safety

SaveXOPPrefsHandle is not thread-safe.

GetXOPPrefsHandle

```
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.

Thread Safety

GetXOPPrefsHandle is not thread-safe.

GetPrefsState

int

GetPrefsState(prefsStatePtr)

int* 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.

Thread Safety

GetPrefsState is not thread-safe.

Programming Utilities

MemClear

```
int
MemClear(void *p, numBytes)
void* p;                // Pointer to memory to be cleared.
BCInt numBytes;         // Number of bytes to be cleared.
```

Sets the specified number of bytes at the memory location pointed to by p to zero.

Thread Safety

MemClear is thread-safe. It can be called from any thread.

GetCStringFromHandle

```
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 null-terminated C string.

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 237.

Thread Safety

GetCStringFromHandle is thread-safe. It can be called from any thread.

PutCStringInHandle

```
int
PutCStringInHandle(str, h)
const char* str;         // Input C string
Handle h;                // Handle to hold text
```

str is a null-terminated C string.

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 237.

Thread Safety

PutCStringInHandle is thread-safe. It can be called from any thread.

CmpStr

```
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.

Thread Safety

CmpStr is thread-safe. It can be called from any thread.

strchr2

```
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.

Thread Safety

strchr2 is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

strrchr2

```
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, `strchr2` is just like `strchr`.

Thread Safety

`strchr2` is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

EscapeSpecialCharacters

```
int
EscapeSpecialCharacters(input, inputLength, output, bufSize, numCharsOutPtr)
const char* input;           // Text to be escaped
int inputLength;             // Number of characters to escape
char* buffer;                // Output buffer
int bufSize;                 // Size of output buffer in characters
int* numCharsOutPtr;         // Output: Number of characters written
```

Converts CR, LF, tab, double-quote and backslash to escape sequences. This is used mostly when you are generating a command to be executed on Igor's command line via **XOPCommand**, **XOPCommand2**, **XOPCommand3** or **XOPSilentCommand** or through Igor's Execute operation. Conversion is necessary because Igor interprets backslash as an escape character.

For example, if you are generating a file loading command and the file uses a Windows UNC path of the form `\\server\share`, the backslashes in the command must be escaped to give `\\\\server\\share` because Igor interprets `\\` as an escape sequence that evaluates to a single backslash.

`input` is a pointer to a string to be escaped. Input may be null-terminated or not.

`inputLength` is the number of characters to be escaped.

NOTE: If `input` is null-terminated and you want output to also be null-terminated, then `inputLength` must include the null terminator character.

`output` is a pointer to a buffer in which escaped output text is returned.

`outputBufferSize` is the total size of the output buffer.

`numCharsOutPtr` is set to the number of characters stored in output. This includes the null-terminator if it was included in `inputLength`.

`input` and `output` can point to the same memory.

Returns 0 if OK or an error code.

Example

```
char str[32];
int length;
// Assume str is set to contain a C string containing characters
// that may need to be escaped.
length = strlen(str) + 1; // Include null-terminator
err = EscapeSpecialCharacters(str, length, str, sizeof(str), &length);
```

Thread Safety

`EscapeSpecialCharacters` is thread-safe. It can be called from any thread.

UnEscapeSpecialCharacters

```
int
UnEscapeSpecialCharacters(input, inputLength, output, outputBufferSize,
numCharsOutPtr)
const char* input;           // Text to be unescaped
int inputLength;             // Number of characters to unescaped
char* buffer;                // Output buffer
int outputBufferSize;        // Size of output buffer in characters
int* numCharsOutPtr;         // Output: Number of characters written
```

Converts escape sequences for CR, LF, tab, double-quote and backslash to CR, LF, tab, double-quote and backslash. You would call this if you receive text that you know is escaped. See **EscapeSpecialCharacters** for a discussion of why text might be escaped.

input is a pointer to a string to be unescaped. Input may be null-terminated or not.

inputLength is the number of characters to be unescaped.

NOTE: If input is null-terminated and you want output to also be null-terminated, then inputLength must include the null terminator character.

output is a pointer to a buffer in which unescaped output text is returned.

outputBufferSize is the total size of the output buffer.

numCharsOutPtr is set to the number of characters stored in output. This includes the null-terminator if it was included in inputLength.

input and output can point to the same memory.

Returns 0 if OK or an error code.

Example

```
char str[32];
int length;
// Assume str is set to contain a C string containing escape sequences.
length = strlen(str) + 1; // Include null-terminator
err = UnEscapeSpecialCharacters(str, length, str, sizeof(str), &length);
```

Thread Safety

UnEscapeSpecialCharacters is thread-safe. It can be called from any thread.

IsINF32

```
int
IsINF32(floatPtr)
float* floatPtr;           // The number to test
```

Returns 1 if the number pointed to by floatPtr is +infinity or -infinity, 0 otherwise.

Thread Safety

IsINF32 is thread-safe. It can be called from any thread.

IsINF64

```
int
IsINF64(doublePtr)
double* doublePtr;         // The number to test
```

Returns 1 if the number pointed to by doublePtr is +infinity or -infinity, 0 otherwise.

Thread Safety

IsINF64 is thread-safe. It can be called from any thread.

IsNaN32

```
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.

Thread Safety

IsNaN32 is thread-safe. It can be called from any thread.

IsNaN64

```
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.

Thread Safety

IsNaN64 is thread-safe. It can be called from any thread.

SetNaN32

```
void
SetNaN32 (floatPtr)
float* floatPtr;
```

Sets the float pointed to by the parameter to NaN (not-a-number).

Thread Safety

SetNaN32 is thread-safe. It can be called from any thread.

SetNaN64

```
void
SetNaN64 (doublePtr)
double* doublePtr;
```

Sets the double pointed to by the parameter to NaN (not-a-number).

Thread Safety

SetNaN64 is thread-safe. It can be called from any thread.

DateToIgorDateInSeconds

```
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.

Thread Safety

DateToIgorDateInSeconds is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

IgorDateInSecondsToDate

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.

Thread Safety

IgorDateInSecondsToDate is Igor-thread-safe with XOP Toolkit 6 and Igor Pro 6.20 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself. See **Igor-Thread-Safe Callbacks** on page 170 for details.

XOPBeep

void

XOPBeep(void)

Emits a beep.

Thread Safety

XOPBeep is not thread-safe.

XOPOKAlert

```
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.

Thread Safety

XOPOKAlert is not thread-safe.

XOPOKCancelAlert

```
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.

Thread Safety

XOPOKCancelAlert is not thread-safe.

XOPYesNoAlert

```
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.

Thread Safety

XOPYesNoAlert is not thread-safe.

XOPYesNoCancelAlert

```
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.

Thread Safety

XOPYesNoCancelAlert is not thread-safe.

MacRectToWinRect

```
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.

Thread Safety

`MacRectToWinRect` is thread-safe. It can be called from any thread.

WinRectToMacRect

```
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.

Thread Safety

`WinRectToMacRect` is thread-safe. It can be called from any thread.

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. This section describes the behavior of the routines as 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 65 for an overview.

NewPtr

```
Ptr
NewPtr (size)
BCInt 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 = (Ptr)NewPtr(100);
if (p == NULL)
    return NOMEM;
```

You must call **DisposePtr** to free the block of memory allocated by NewPtr.

In XOP Toolkit 6, the prototype for NewPtr on Windows was changed. See **Changes To NewPtr and DisposePtr** on page 448 for details.

Thread Safety

NewPtr is thread-safe. It can be called from any thread.

GetPtrSize

```
BCInt
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.

Thread Safety

GetPtrSize is thread-safe. It can be called from any thread.

SetPtrSize

```
void  
SetPtrSize(p, size)  
Ptr p;  
BCInt 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 succeeded.

Thread Safety

SetPtrSize is thread-safe. It can be called from any thread. However, you can not call MemError to determine if an error occurred from a preemptive thread. You can call GetPtrSize after calling SetPtrSize to determine if SetPtrSize succeeded.

DisposePtr

```
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.

In XOP Toolkit 6, the prototype for DisposePtr on Windows was changed. See **Changes To NewPtr and DisposePtr** on page 448 for details.

Thread Safety

DisposePtr is thread-safe. It can be called from any thread.

NewHandle

```
Handle  
NewHandle(size)  
BCInt 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

Thread Safety

NewHandle is thread-safe. It can be called from any thread.

HandToHand

```
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 succeeded 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);
```

Thread Safety

HandToHand is thread-safe. It can be called from any thread.

HandAndHand

```
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 succeeded or an error code.

Thread Safety

HandAndHand is thread-safe. It can be called from any thread.

GetHandleSize

```
BCInt
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.

NOTE: In Igor Pro 7 and later wave handles are not regular handles. In the unlikely event that you need to know the size of a wave handle, use **WaveMemorySize**, not GetHandleSize.

Thread Safety

GetHandleSize is thread-safe. It can be called from any thread.

SetHandleSize

```
void
SetHandleSize(h, size)
Handle h;
BCInt 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.

After increasing the size of a handle, you must re-dereference it. See **Resizing Handles** on page 199 for details.

SetHandleSize can fail. You must call **MemError** after calling SetHandleSize to verify that the resizing succeeded.

Example

```
Handle h;
Ptr p;
int err = 0;

h = NewHandle(100);           // Allocate handle
if (h == NULL)
    return NOMEM;
p = *h;                       // Dereference handle
. . .                         // Work with handle data using p
SetHandleSize(h, 200);        // This may relocate handle data in memory
if (err = MemError()) {
    DisposeHandle(h);
    return err;
}
p = *h;                       // Re-dereference handle
. . .                         // Work with handle data using p
DisposeHandle(h);             // All done
```

Thread Safety

SetHandleSize is thread-safe. It can be called from any thread. However, you can not call MemError to determine if an error occurred from a preemptive thread. You can call GetHandleSize after calling SetHandleSize to determine if SetHandleSize succeeded.

DisposeHandle

```
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.

Thread Safety

DisposeHandle is thread-safe. It can be called from any thread.

PtrToHand

```
int  
PtrToHand(srcPtr, destHandlePtr, size)  
Ptr srcPtr;  
Handle* destHandlePtr;  
BCInt 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 succeeded or an error code.

Thread Safety

PtrToHand is thread-safe. It can be called from any thread.

PtrAndHand

```
int  
PtrAndHand(p, h, size)  
Ptr* p;  
Handle h;  
BCInt 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.

After increasing the size of a handle you must re-dereference it. See **Resizing Handles** on page 199 for details.

PtrAndHand returns 0 if the allocation succeeded or an error code.

Thread Safety

PtrAndHand is thread-safe. It can be called from any thread.

MemError

```
int  
MemError(void)
```

MemError returns the error status from the most recently called memory management routine.

Use MemError after calling Macintosh memory management routines, such as **SetPtrSize** and **SetHandleSize**, which do not directly return an error code. See **Resizing Handles** on page 199 for further discussion.

Thread Safety

MemError is not thread-safe.

Emulated Menu Management Routines

Prior to XOP Toolkit 6.40 Windows XOPs used emulations of Macintosh Menu Manager routines provided through IGOR.lib. As of XOP Toolkit 6.40, these emulated routines are no longer supported. New XOPs and existing XOPs that you are updating must be revised to use the new routines. They are documented under **Routines for XOPs with Menu Items** on page 314.

See Chapter 8, **Adding Menus and Menu Items**, for details on menus.

Miscellaneous Emulated Macintosh Routines

TickCount

TickCountInt

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.

Thread Safety

TickCount is thread-safe. It can be called from any thread.

XOP Toolkit 6 Upgrade Notes

Overview

These notes are for XOP programmers transitioning from XOP Toolkit 5 to XOP Toolkit 6.40.

This version of the manual describes XOP development with Xcode 6 on Macintosh and Visual C++ 2015 on Windows. For details on older development systems, download the XOP Toolkit 6.30 manual from:
<ftp://ftp.wavemetrics.net/IgorXOPToolkit/Manual/Old/XOPToolkitManual630.pdf>

Here are the main new aspects of XOP Toolkit 6:

- Supports development of 64-bit XOPs on Windows
- Supports use of Xcode 6 on Macintosh
- Supports use of Visual C++ 2015 on Windows
- No longer supports old development systems
- Better support for multithreading (requires Igor Pro 6.20 or later)
- No longer supports Igor Pro 5 and earlier
- Obsolete XOPSupport routines have been removed
- All sample XOP source files were converted from C to C++

XOP Toolkit 6 can create XOPs compatible with Igor Pro 6 or later. If you must support earlier versions of Igor then you must continue to use XOP Toolkit 5.

In XOP Toolkit 6.30 and later the sample XOP projects are configured such that they require Igor Pro 6.20 or later.

If you use the old (pre-Igor Pro 5) method of creating external operations you will need to either rewrite your operations to use **Operation Handler** or continue to use XOP Toolkit 5.

If you are updating an existing XOP, please familiarize yourself with the entirety of this appendix. Most of the changes will not affect you but you should know what they are just in case.

Development Systems Supported by XOP Toolkit 6

XOP Toolkit 6.40 provides support and samples for creating XOPs using these development systems:

For Mac OS XOPs: Apple's Xcode 6. Later versions may work.

For Windows XOPs: Microsoft's Visual C++ 2015. Later versions may work.

Sample projects are included for older development systems but this version of the manual does not discuss using them.

Create a Copy of Your Existing XOPs

The support and sample files for XOP Toolkit 6 come in a folder named IgorXOPs6.

If you have existing XOP projects, *copy* them into the IgorXOPs6 folder. Leave your old projects as they are and do all new development in the new folder.

Changes You Must Make

If you have XOPs developed with XOP Toolkit 5, here are the changes that you must make in order to use XOP Toolkit 6. We recommend that you familiarize yourself with this entire appendix first and then come back to this section and perform each of these steps.

Update your Macintosh project as described under **Xcode Project Changes For XOP Toolkit 6** on page 457.

Update your Windows project as described under **Visual C++ Project Changes For XOP Toolkit 6** on page 460 and **Modifying Your VC2015 Project For XOP Toolkit 6** on page 461.

Change your main function to XOPMain as described under **Changing the main Function** on page 451.

XOP Toolkit 6.40 requires Igor Pro 6.20 or later. Make sure that your XOPMain function checks the version of Igor it is running with. See **Checking the Igor Version** on page 452 below for details.

Replace all invocations of "XOPStructureAlignmentTwoByte.h" and "XOPStructureAlignmentReset.h" with "#pragma pack" statements. See **Structure Alignment in XOP Toolkit 6** on page 443 below for details.

Replace "#ifdef _MACINTOSH_" with "#ifdef MACIGOR".

Replace "#ifdef _WINDOWS_" with "#ifdef WINIGOR".

Replace DOUBLE (all caps) with double.

If you use the DataObjectValue structure, its wavH field has been renamed waveH; change your code accordingly.

Remove all calls to MoveLockHandle, MoveHHi, HLock, HGetState, HSetState and HUnlock from your source code. These calls are all obsolete. See **Remove Handle Locking Code** on page 452 for details.

Obsolete XOPSupport routines have been removed from XOP Toolkit 6. See the section **Removed XOPSupport Routines** on page 445 for the list of removed routines. Only very old XOPs use these routines. If you use them and do not want to modify your XOP's source code then you should continue to use XOP Toolkit 5.

Among the removed routines are obsolete command parsing routines used by XOPs developed prior to Igor Pro 5. If your XOP adds an operation using these obsolete routines, see **Removed Command Parsing Routines** on page 447.

If you use **XOPGetFilePosition2**, **XOPSetFilePosition2** or **XOPNumberOfBytesInFile2** you must change your code. These routines now take SInt64 parameters instead of doubles.

Some XOPSupport parameter types have changed for 64-bit compatibility. For example, some longs were changed to ints. These changes may affect you even if you are not compiling your XOP for 64 bits. If there is a problem, you will get a compiler warning or error. To familiarize yourself with this issue, read **64-bit XOPs** on page 453 and especially the subsection **Changes For 64-bit Compatibility** on page 453.

Build and test your XOP with XOP Toolkit 6. Examine all compiler warnings and determine if they are harmless or indicate a real problem. Eliminate compiler warnings where possible.

If you want your XOP to run with the 64-bit version of Igor, see **64-bit XOPs** on page 453.

New Features

XOP Toolkit 6 has better support for multithreading. See **Multithreading** on page 167.

XOP Toolkit 6 supports 64-bit XOP development. See **64-bit XOPs** on page 181.

You can define external functions that return wave references. See **Wave Results** on page 103.

You can define external functions that return data folder references (see **Data Folder Reference Results** on page 104) and take data folder references as parameters (see **Data Folder Reference Parameters** on page 104).

You can define external operations that take data folder references as parameters. See **Data Folder Reference Parameters** on page 84.

All sample XOP source files in XOP Toolkit 6 are C++ instead of C files. For details see **Sample XOP Source Files Converted To C++** on page 454.

XOPStandardHeaders Changes

XOPStandardHeaders.h now creates the following preprocessor macros which can be used for conditional compilation:

Macro	Description
MACIGOR	Defined if compiling for Macintosh.
WINIGOR	Defined if compiling for Windows.
IGOR32	Defined if compiling a 32-bit executable.
IGOR64	Defined if compiling a 64-bit executable.

Replace occurrences of `_MACINTOSH_` with `MACIGOR` in your source files.

Replace occurrences of `_WINDOWS_` with `WINIGOR` in your source files.

Previous versions of the XOP Toolkit required defining the symbol `TARGET_RT_MAC_MACHO` for the Macintosh Rez compiler. Starting with XOP Toolkit 6.00, this is no longer necessary.

Organizational Changes

The `XOPFiles.c` file has been split into three files: `XOPFiles.c`, `XOPFilesMac.c` and `XOPFilesWin.c`.

Igor error codes were moved from `IgorXOP.h` to a new file, `IgorErrors.h`, in the `XOPSupport` folder. `IgorErrors.h` is `#included` in `XOPStandardHeaders.h` so if you `#include XOPStandardHeaders.h` you do not need to `#include IgorErrors.h`.

Structure Alignment in XOP Toolkit 6

External function and external operation parameter structures are passed from Igor to your XOP. Igor uses two-byte alignment for structures passed to or received from XOPs.

To ensure this alignment, you need to use pragmas to tell the compiler to align your external function and external operation parameter structures on two-byte boundaries. Otherwise a crash will occur.

In XOP Toolkit 5, this was done using `#include` statements:

```
#include "XOPStructureAlignmentTwoByte.h"
```

Appendix A — XOP Toolkit 6 Upgrade Notes

```
. . . structure definitions here . . .
```

```
#include "XOPStructureAlignmentReset.h"
```

The files `XOPStructureAlignmentTwoByte.h` and `XOPStructureAlignmentReset.h` were provided by XOP Toolkit 5 and contained compiler directives. Different development systems required different directives for structure packing and these files applied the right directives for a given compiler.

Now only Xcode and Visual C++ development systems are supported for XOP development. Both of these support the `pack` pragma. Consequently, XOP Toolkit 6 uses a simpler technique:

```
#pragma pack(2)          // All structures passed to Igor must use 2-byte alignment
```

```
. . . structure definitions here . . .
```

```
#pragma pack()           // Return to default structure packing
```

The files `XOPStructureAlignmentTwoByte.h` and `XOPStructureAlignmentReset.h` have been removed from XOP Toolkit 6 so you must replace references to them with `#pragma pack` statements.

Search your source code and replace this:

```
#include "XOPStructureAlignmentTwoByte.h"
```

with this:

```
#pragma pack(2)          // All structures passed to Igor must use 2-byte alignment
```

Search again and replace this:

```
#include "XOPStructureAlignmentReset.h"
```

with this:

```
#pragma pack()           // Return to default structure packing
```

NOTE: Be careful to use the right `#pragma pack` statement. If you use the wrong statement, your code will crash in mysterious ways. Verify that each structure shared with Igor has `"#pragma pack(2)"` before the structure definition and `"#pragma pack()"` after the structure definition.

These `pack` pragmas are needed only around the *definition* of a structure that is passed between Igor and the XOP. They are not needed where the structure is merely *used*.

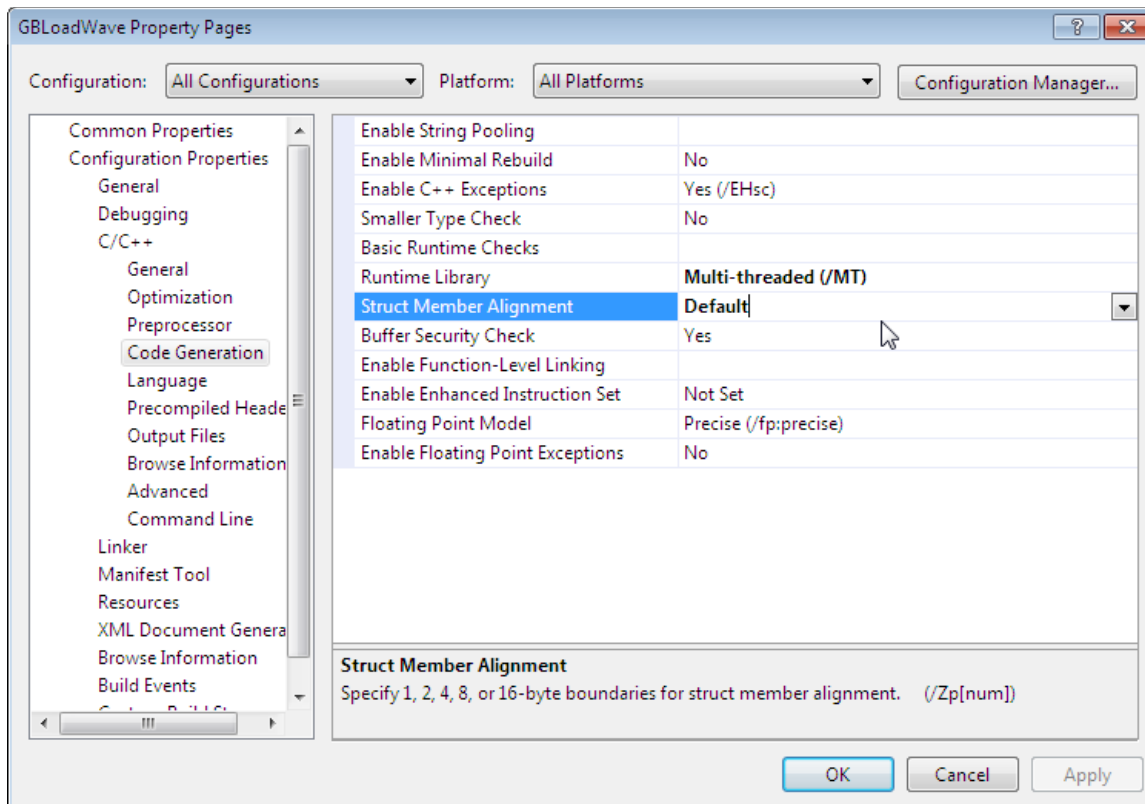
These `pack` pragmas are also not needed, and for efficiency are not recommended, for structures used just within your XOP.

Visual C++ Struct Member Alignment Setting

Visual C++ has a "Struct Member Alignment" setting. This setting is found in the Code Generation section of the C/C++ category of the project properties window. The setting specifies the maximum number of padding bytes that the compiler may insert in a structure to align fields on their natural boundaries.

In XOP Toolkit 5 and earlier, the recommendation was to set the Struct Member Alignment setting to two-byte alignment. The reason for this was to guarantee that external function and external operation parameter structures, which are passed from Igor to your function, were two-byte aligned, as Igor requires. This recommendation was made to guarantee the right structure alignment in the event that you forgot to specify structure alignment using the technique described in the preceding section.

Setting Struct Member Alignment to two bytes can cause problems in code that calls libraries whose header files assume default structure alignment. Therefore, in XOP Toolkit 6, the recommended Struct Member Alignment setting has been changed from two bytes to "default":



Microsoft documentation says that "default" equates to 8-byte alignment:

<http://msdn.microsoft.com/en-us/library/xh3e3fd0.aspx>

However, empirically, "default" gives you 8-byte struct member alignment in Win32 and 16-byte struct member alignment in x64. We determined this using "#pragma pack(show)" which emits a warning specifying the struct member alignment in effect.

In any event, default gives the alignment used by the Windows header files and also by most libraries that you might use.

This change, which is reflected in the sample XOP projects, makes it imperative that you specify the structure alignment for structures shared with Igor, as described in the preceding section.

A technical discussion of Windows data alignment can be found here:

[http://msdn.microsoft.com/en-us/library/aa290049\(VS.71\).aspx](http://msdn.microsoft.com/en-us/library/aa290049(VS.71).aspx)

Removed XOPSupport Routines

The following sections list routines removed from XOP Toolkit 6 and replacement routines, if any.

Removed General Routines

The following routines have been removed from the XOP Toolkit:

Removed Routine	Replacement Routine	Notes
SetXOPRefCon	None	Use a global variable instead.
GetXOPRefCon	None	Use a global variable instead.

Appendix A — XOP Toolkit 6 Upgrade Notes

Removed Routine	Replacement Routine	Notes
Capitalize	None	Use CmpStr instead of calling Capitalize and strcmp.
GetDLong (used in dialogs)	GetDInt or GetDInt64	Use GetDInt to get a signed 32-bit quantity. Use GetDInt64 to get a signed 64-bit quantity.
SetDLong (used in dialogs)	SetDInt or SetDInt64	Use SetDInt with a signed 32-bit quantity. Use SetDInt64 with a signed 64-bit quantity.
MoveLockHandle MoveHHi HLock / HUnlock HGetState / HSetState	None	Remove all calls to MoveLockHandle, MoveHHi, HLock, HUnlock, HGetState and HSetState from your source code. These calls are all obsolete.
DoCHIO	None	DoCHIO was used for character-oriented I/O for the old VDT and old NIGPIB XOPs. See VDT2 and NIGPIB2 for substitute techniques. See CHIO Routines Removed on page 447 for details.
GetPathInfo	GetPathInfo2	
GetFullMacPathToDirectory	None	
IgorVersion	None	Use the igorVersion global instead.
IsMacOSX	None	Use #ifdef MACIGOR.
WaveHandlesModified	None	Use WaveHandleModified in a loop.
GetWavesInfo	None	Use WaveType, WavePoints and WaveData as needed.
SetWavesStates	None	No replacement is needed.

Removed Text Utility Routines

The following text utility routines have been removed from the XOP Toolkit.

Removed Routine	Replacement Routine	Notes
TUNew	TUNew2	
TUSetSelect	TUSetSelLocs	
TUGetText	TUFetchParagraphText	
TUFetchText	TUFetchParagraphText	
TULength	TUGetDocInfo	
TUSelStart	TUGetSelLocs	
TUSelEnd	TUGetSelLocs	
TUSelectionLength	TUGetSelLocs	

Removed Routine	Replacement Routine	Notes
TUInsertFile	None	You can read a file's contents using XOPReadFile and then insert using TUInsert.
TUWriteFile	None	You can get a text window's text using TUSelectAll and TUFetchSelectedText and then write using XOPWriteFile.

Removed Command Parsing Routines

The following command parsing routines have been removed from XOP Toolkit 6. These routines were used by XOPs created for Igor Pro 4 and earlier. XOPs created starting with Igor Pro 5 use **Operation Handler** described in Chapter 5.

NextSymb, GetSymb, IsStringExpression
 GetFlag, GetFlagNum, GetTrueOrFalseFlag
 GetAString, GetAStringInHandle, GetFormat
 GetName, Keyword, GetKeyword
 GetNum, GetNum2, GetLong
 GetWaveName, GetWave, GetWaveList, GetWaveRange
 GetNumVarName, GetStrVarName
 GetDataFolder, GetDataFolderAndName
 AtEndOfCommand, CheckTerm

If you use these routines you must either rewrite your operation using Operation Handler or you must continue to use XOP Toolkit 5.

Although these routines have been removed from the XOP Toolkit, Igor Pro 6 still supports them for the benefit of old XOPs. However, they may be dropped in a future version of Igor.

CHIO Routines Removed

The DoCHIO ("do character-oriented I/O") XOPSupport routine was removed in XOP Toolkit 6. Also removed were the supporting CHIORec structure and various macros (CHIOREAD, CHIOWRITE, ...).

DoCHIO was used in the old VDT and old NIGPIB XOPs to parse received numbers and strings. The DoCHIO routine relied on callbacks provided by the XOP to get the characters but did the parsing itself. This was too complex and bug-prone. As a result, in VDT2 and NIGPIB2, DoCHIO was replaced by simple routines defined in the XOPs themselves.

Although these routines have been removed as of the XOP Toolkit 6, Igor Pro 6 still supports them for the benefit of old XOPs. However, they may be dropped in a future version of Igor.

If you have an XOP that uses DoCHIO, you must either change your XOP or continue to use XOP Toolkit 5.

Renamed XOPSupport Routines

Routines in XOPNumericConversion.c which had "Long" in their names were renamed to use "SInt32". Routines which had "UnsignedLong" in their names were renamed to use "UInt32". Most XOPs do not use these routines and therefore require no changes.

Changed XOPSupport Routines

If you use **XOPGetFilePosition2**, **XOPSetFilePosition2** or **XOPNumberOfBytesInFile2** you must change your code. These routines now take SInt64 parameters instead of doubles. The previous versions of this routines were available only in a beta version of XOP Toolkit 5.09.

Appendix A — XOP Toolkit 6 Upgrade Notes

The XOPNavOpenFileDialog and XOPNavSaveFileDialog routines were moved from XOPSupportMac.c to XOPFilesMac.c and made static and changed to support long file names. You should not use these routines. Use the the platform-independent **XOPOpenFileDialog** and **XOPSaveFileDialog** routines.

The MAX_FILENAME_LEN and MAX_DIRNAME_LEN macros were changed to be 255 characters on Macintosh.

The following routines now support long file names on Mac OS X: **XOPCreateFile**, **XOPDeleteFile**, **XOPOpenFile**, **XOPOpenFileDialog**, **XOPSaveFileDialog**. If you do all of your file handling through these and other XOPSupport routines then all you need to do is recompile your XOP and it should work with long file names on Macintosh.

On Macintosh, XOPOpenFileDialog and XOPSaveFileDialog can return path elements that exceed 31 characters in length. However, they still return paths using HFS syntax. If a path element exceeds 31 characters in length, you can use the path with XOPCreateFile, XOPDeleteFile and XOPOpenFile but to use it with other routines you need to convert it to another format such as Posix using the HFSToPosixPath routine.

On Windows the Open File dialog displayed by **XOPOpenFileDialog** and the Save File dialog displayed by **XOPSaveFileDialog** are now resizable.

Changes To NewPtr and DisposePtr

For historical reasons, in XOP Toolkit 5, the prototypes for the **NewPtr** and **DisposePtr** routines were different on Windows and Macintosh:

XOP Toolkit 5 Macintosh	XOP Toolkit 5 Windows
Ptr NewPtr(Size);	void* NewPtr(long);
void DisposePtr(Ptr);	void DisposePtr(void*)

In XOP Toolkit 6, the prototypes have been changed to match the Macintosh prototypes:

XOP Toolkit 6 Macintosh	XOP Toolkit 6 Windows
Ptr NewPtr(Size);	Ptr NewPtr(BCInt);
void DisposePtr(Ptr);	void DisposePtr(Ptr)

You may need to add casts to prevent compiler errors in existing code:

```
int* p;  
p = (int*)NewPtr(100);
```

New XOPSupport Routines

The following sections list new routines added in XOP Toolkit 6.

Many of these routines were also added in XOP Toolkit 5.09 which was released at the same time as XOP Toolkit 6. XOP Toolkit 5.09 is for XOP programmers who can not use XOP Toolkit 6 because they are stuck with old development systems.

New File Routines

The following file routines were added in XOP Toolkit 6.

New Routine	Notes
XOPReadFile64	Use this to read more than 4 GB in one call when running as 64 bits.
XOPWriteFile64	Use this to write more than 4 GB in one call when running as 64 bits.

New Routine	Notes
XOPGetFilePosition2	Gets the current file position as a 64-bit number.
XOPSetFilePosition2	Sets the current file position using a 64-bit number.
XOPNumberOfBytesInFile2	Returns the number of bytes in the file as a 64-bit number.
ParseFilePath	Returns parts of file paths.
SpecialDirPath	Returns paths to special directories such as preference and temporary directories.

New Wave and Data Folder Routines

The following wave and data folder routines were added in XOP Toolkit 6.

New Routine	Notes
HoldWave	Supports wave reference counting.
ReleaseWave	Supports wave reference counting.
HoldDataFolder	Supports data folder reference counting.
ReleaseDataFolder	Supports data folder reference counting.

New Operation Handler Routines

The following Operation Handler-related routines were added in XOP Toolkit 6.

New Routine	Notes
GetOperationWaveRef	Supports destination wave in operations. Usually GetDestinationWave is used instead.
GetOperationDestWave	Handles creation of destination wave in operations.
SetOperationStrVar2	Stores text or binary data in an external operation output variable.
FetchNumericDataUsingVarName	See VarName Parameters on page 83 for details.
FetchStringDataUsingVarName	See VarName Parameters on page 83 for details.

New Miscellaneous Routines

The following miscellaneous routines were added in XOP Toolkit 6.

New Routine	Notes
RunningInMainThread	Returns the truth that you are running in the main thread.
CheckRunningInMainThread	Returns the truth that you are running in the main thread. If not, an error dialog is displayed.
GetDInt64	Returns a value entered in a dialog editText field as a 64-bit integer.
SetDInt64	Sets a dialog editText field to a 64-bit integer value.
TUFetchText2	Returns text from a section of a TU document.
XOPNotice3	This is useful only in rare situations and only with the “History Carbon Copy” feature added to Igor Pro 6.10B04.

Appendix A — XOP Toolkit 6 Upgrade Notes

New Routine	Notes
EscapeSpecialCharacters	For dealing with CR, LF, tab, double-quote and backslash in strings.
UnEscapeSpecialCharacters	For dealing with CR, LF, tab, double-quote and backslash in strings.

Miscellaneous Changes

Prior to XOP Toolkit 6, if the numPoints parameter to **MakeWave** or **FileLoaderMakeWave** was zero, the wave was created with 128 points. This behavior was a relic of a time when Igor required that a wave contain at least two points (prior to Igor Pro 3.0). In XOP Toolkit 6, when numPoints is zero, a zero-point wave is created.

A subtle change was made to **CreateValidDataObjectName**. If the proposed name, in the inName parameter, is in conflict with another type of object or with a function or operation name, and if allowOverwrite is non-zero, the output name generated is now allowed to be the same as another object of the specified type. Previously in this situation, the output name would always be unique. For example, if the input name is datetime, which conflicts with the Igor datetime function, and allowOverwrite is non-zero, the output name will now be datetime0 even if a datetime0 already exists so long as it is of the same object type. Previously the output name in this situation would be made unique, for example, datetime1. The new behavior is more in keeping with the spirit of the allowOverwrite parameter which will be set if the caller wants to overwrite an existing object of the same name.

Fixed a bug in **CreateValidDataObjectName**. It did not work correctly for data folder objects.

Some char* XOPSupport parameters were changed to const char* (e.g., **GetNamedDataFolder**).

Added the following preprocessor macros to the XOPSupport projects on Windows:

```
_WIN32_WINNT=0x0500
WINVER=0x0500
```

This triggers some more modern behaviors such as enabling the showing of the “Places” pane in the Open File dialog.

Spelling of the following structure names was corrected:

```
FIFOFileHeader (was FIFOFIleHeader)
FIFOSplitFileHeader (was FIFOSplitFIleHeader)
```

The DataObjectValue structure’s wavH field was renamed waveH. This structure is defined in IgorXOP.h.

DOUBLE was removed from IgorXOP.h. If you use it, use double instead.

CtoToolStr and TooltoCStr were removed from IgorXOP.h. They should not be needed.

MAXWAVES was removed from IgorXOP.h. It should not be needed.

WNAME_SIZE was removed from IgorXOP.h. It should not be needed.

The following macros were removed from IgorXOP.h:

```
#define F32_NAN 0x7fffffffL
#define F32_PLUS_INF 0x7f800000L
#define F32_MINUS_INF 0xff800000L
#define F32_SIGN_MASK 0x80000000L
#define F32_NOSIGN_MASK 0x7fffffffL
#define F32_INF 0x7f800000L // 32 bit infinity
#define F64_INF 0x7ff00000L // 64 bit infinity
```



```
#define LEFT_ARROW_CHAR_CODE
#define RIGHT_ARROW_CHAR_CODE 0x1D
#define UP_ARROW_CHAR_CODE 0x1E
#define DOWN_ARROW_CHAR_CODE 0x1F
#define PAGE_UP_CHAR_CODE 0x0B
#define PAGE_DOWN_CHAR_CODE 0x0C
#define HOME_CHAR_CODE 0x01
#define END_CHAR_CODE 0x04
#define FORWARD_DELETE_CHAR_CODE 0x7F
#define HELP_CHAR_CODE 0x05

#define LEFT_ARROW_KEY_CODE 0x7B
#define RIGHT_ARROW_KEY_CODE 0x7C
#define UP_ARROW_KEY_CODE 0x7E
#define DOWN_ARROW_KEY_CODE 0x7D
#define PAGE_UP_KEY_CODE 0x74
#define PAGE_DOWN_KEY_CODE 0x79
#define HOME_KEY_CODE 0x73
#define END_KEY_CODE 0x77
#define FORWARD_DELETE_KEY_CODE 0x75
#define HELP_KEY_CODE 0x72

#define MODIFIER_KEYS_MASK (cmdKey | optionKey | controlKey | shiftKey)

#define TUMAXBYTES 32000
```

All hmnu (menu) and hdlg (dialog) balloon help resources were removed from Macintosh .r files. Mac OS X does not support balloon help.

Changed “afxres.h” to “winresrc.h” in all .rc files for compatibility with Visual C++ 2010 Express Edition.

Changing the main Function

In XOP Toolkit 5, the main XOP function was called “main” and had this form:

```
HOST_IMPORT void          // *** Was void
main(IORecHandle ioRecHandle) // Definition in .cpp file
{
    ...
}
```

Modern compilers no longer allow this signature for main. Consequently, in XOP Toolkit 6, you must change your main function to this:

```
HOST_IMPORT int
XOPMain(IORecHandle ioRecHandle) // XOPMain requires Igor Pro 6.20 or later
{
    XOPInit(ioRecHandle);
    SetXOPEntry(XOPEntry);

    if (igorVersion < 620) {
        SetXOPResult(IGOR_OBSOLETE);
        return EXIT_FAILURE;
    }

    // Do your custom initialization here

    SetXOPResult(0);
    return EXIT_SUCCESS;
}
```

Remember also to change the prototype for your main function in your .h file.

Checking the Igor Version

XOP Toolkit 6.40 requires Igor Pro 6.20 or later. The main reason for this is the restriction imposed by modern compilers as explained under **Changing the main Function** on page 451. Secondary reasons include changes to support multi-threading, wave references and data folder references.

To prevent your XOP from running with earlier versions of Igor, you must include a test in your XOPMain function, like this:

```
if (igorVersion < 620) {  
    SetXOPResult(IGOR_OBSOLETE);  
    return EXIT_FAILURE;  
}
```

IGOR_OBSOLETE is a built-in Igor error code that generates an error alert that says "XOP requires a later version of Igor".

Remove Handle Locking Code

In the XOP Toolkit, Handles are used for waves, data folders and string values. The Handle was created for the original Mac OS as a way to allocate memory. On Windows the XOP Toolkit emulates the Macintosh handle-related routines so that that the same code works on Macintosh and Windows.

A handle is a pointer to a pointer to a block of memory allocated in the heap. To access the block of memory, the application must "dereference" the handle to produce a pointer:

```
Handle h = NewHandle(2);    // Allocate a block of 2 bytes  
char* p = *h;              // Dereference h to obtain a pointer  
p[0] = 0;                  // Access the block of memory  
p[1] = 1;
```

To prevent memory fragmentation, the original Mac OS operating system was allowed to move the block of memory anytime any memory allocation was done through a call to NewHandle, SetHandleSize, NewPtr or SetPtrSize. Because of this, if you dereferenced a handle you had to "lock" it if there was any chance that a memory allocation might be done while you were using the resulting pointer. Locking a handle told the operating system that it could not move the block of memory.

Mac OS X, unlike the original Mac OS, does not move blocks of memory any time any allocation is done. The XOP Toolkit emulation of handles also does not do this. Since Igor no longer runs on Mac OS 9, it is no longer necessary to lock handles. Consequently, calls to the Mac OS routines MoveHHI, HLock, HUnlock, HGetState and HSetState should be removed from XOP source code. On Windows, using XOP Toolkit 6, they *must* be removed because the emulation of these routines has been removed from the XOP Toolkit. Also, calls to MoveLockHandle must be removed on both Macintosh and Windows as this routine has been removed from XOP Toolkit 6.

NOTE: The block of memory referenced by a handle can still be relocated if you change the size of the block. This is true whether you increase or decrease its size. See **Resizing Handles** on page 199 for details.

Callbacks From Igor Threads

In XOP Toolkit 5 you could create thread-safe external operations and functions but they were forbidden from doing any callbacks to Igor when running in a pre-emptive thread. In XOP Toolkit 6 when running with Igor Pro 6.20 or later, this restriction is relaxed. See **Multithreading** on page 167 for details.

64-bit XOPs

For general information on programming 64-bit XOPs, see **64-bit XOPs** on page 181.

The next section explains changes in XOP Toolkit 6 to support 64-bit XOPs.

Changes For 64-bit Compatibility

This section discusses changes in XOP Toolkit 6 to enable the creation of 64-bit XOPs.

Removed XOPSupport Routines

The `GetDLong` and `SetDLong` routines have been removed. Use `GetDInt/SetDInt` with 32-bit values and `GetDInt64/SetDInt64` with 64-bit values.

Changed XOPSupport Routine Parameters

All longs used in XOPSupport routines have been changed to ints or to the appropriate WaveMetrics data type.

This affects the prototypes for many XOPSupport routines. Most of these changes have no impact on your source code. For example, the `SetDataFolderProperties` XOPSupport routine was changed from:

```
int SetDataFolderProperties(DataFolderHandle, long);
```

to

```
int SetDataFolderProperties(DataFolderHandle, int);
```

No change to your source code is required.

However, the `GetDataFolderProperties` was changed from:

```
int GetDataFolderProperties(DataFolderHandle, long*);
```

to

```
int GetDataFolderProperties(DataFolderHandle, int*);
```

If you call this routine passing a pointer to a long you will get a compile warning or error. You need to pass a pointer to an int instead. For example, change this:

```
long properties;
result = GetDataFolderProperties(dataFolderH, &properties);
```

to this:

```
int properties;
result = GetDataFolderProperties(dataFolderH, &properties);
```

The easiest way to find where such changes are necessary is to compile and then fix all warnings and errors.

The `MDGetWaveDimensions` callback is a commonly-used case where this kind of change is needed. Change this:

```
long numDimensions;
long dimensionSizes[MAX_DIMENSIONS+1];
result = MDGetWaveDimensions(waveH, &numDimensions, dimensionSizes);
```

to this:

```
int numDimensions;
CountInt dimensionSizes[MAX_DIMENSIONS+1];
result = MDGetWaveDimensions(waveH, &numDimensions, dimensionSizes);
```

Sample XOP Source Files Converted To C++

For XOP Toolkit 6, all sample source files were converted from C to C++. The programming techniques in the sample XOPs are still C techniques. The XOPSupport library remains C.

In case you want to convert a C XOP to C++, here are the steps we took.

1-Xcode 3.

Open the Xcode project in Xcode 3, right-click each .c file, choose Rename, and change the extension to .cpp.

When you rename a .c file as .cpp in Xcode, Xcode changes the File Type setting for the file from “sourcecode.c.c” to “sourcecode.cpp.cpp”. You can see this in the General tab of the file’s Info window.

In the Targets group, double-click the XOP’s target icon to display the Target Info window.

Click the Build tab.

For all configurations, in the GCC Warnings section, remove anything in the Other Warning Flags setting.

1-Visual C++.

Open the Visual C++ project, right-click each .c file in the Solution Explorer window, choose Rename, and change the extension to .cpp.

If you are using the same files for Xcode and Visual C++ then you already renamed the file in step 1-Xcode and you will get an error in this step. In that case, do this instead:

Choose View→Other Windows→Properties Window (or press Alt-Enter). Now click each .c file in the Solution Explorer window and change its extension in the Properties window to .cpp.

Choose File→Save All.

2. Open the main .cpp file, find the XOPEntry function, and change the linkage declaration from static to extern “C”.

This change is appropriate because Igor calls XOPEntry using C calling conventions.

For example, change this:

```
static void      // C code
XOPEntry(void)
```

to this:

```
extern "C" void  // C++ code
XOPEntry(void)
```

3. For each external function execute routine, change the linkage declaration from static to extern “C”.

This change is necessary because Igor calls these routines using C calling conventions.

For example, in XFUNC1 we changed this:

```
static int
XFUNC1Add(XFUNC1AddParams* p)
```

to this:

```
extern "C" int
XFUNC1Add(XFUNC1AddParams* p)
```

A good way to find such routines, assuming you followed the pattern of the sample XOPs, is to search for “Params* p” or “ParamsPtr p”.

4. For each external operation execute routine, change the linkage declaration from static to extern “C”.

This change is necessary because Igor calls these routines using C calling conventions.

For example, in XOP1 we changed this:

```
static int
ExecuteXOP1 (XOP1RuntimeParamsPtr p)
```

to this:

```
extern "C" int
ExecuteXOP1 (XOP1RuntimeParamsPtr p)
```

A good way to find such routines, assuming you followed the pattern of the sample XOPs, is to search for “ParamsPtr p” or “Params* p”.

5. For each external operation, add a void* cast to the fifth parameter in the RegisterOperation call.

This change is necessary because C++ is pickier than C about pointer types.

For example, in XOP1 we changed this:

```
return RegisterOperation(..., ExecuteXOP1, 0);
```

to this:

```
return RegisterOperation(..., (void*)ExecuteXOP1, 0);
```

6. If you call external functions using the message method instead of the direct method, add a cast of the p parameter.

Most XOPs do not use this technique.

This change is necessary because C++ is pickier than C about pointer types.

For example, in XFUNC1, in the DoFunction routine, we changed this:

```
err = XFUNC1Add(p);
```

to this:

```
err = XFUNC1Add((XFUNC1AddParams*)p);
```

7. If you call the WaveData XOPSupport routine, add a cast to its return value.

This change is necessary because C++ is pickier than C about pointer types.

For example, in XFUNC2 we changed this:

```
fPtr = WaveData(p->waveHandle);
```

to this:

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

8. Compile your XOP.

You will probably get some errors where C++ is pickier than C. For example, in many cases we had to change this:

```
char* p = malloc(numBytes);
```

to this:

```
char* p = (char*)malloc(numBytes);
```

because malloc returns a void* and C++ does not allow us to assign a void* to a char*.

9. Fix any errors you got in the previous step and repeat.

Repeat steps 8 and 9 until you get no more errors.

10. Fix any warnings that you can fix.

You may get warnings. Examine each warning and fix those that you can fix.

You may get warnings that you can't fix. For example, in Xcode, we get this warning:

warning: object file compiled with -mlong-branch which is no longer needed.
To remove this warning, recompile without -mlong-branch: crt3.o

This can't be fixed because it requires recompiling an Apple C runtime library file.

Appendix A — XOP Toolkit 6 Upgrade Notes

In addition to the steps explained above, we changed the definition of `HOST_IMPORT` on Macintosh so that when used in a C++ file, it includes `extern "C"`. This change was made in the `XOPStandardHeaders.h` file. `HOST_IMPORT` is used to declare the main XOP function which is called by Igor using C calling conventions.

Project Changes For XOP Toolkit 6

Overview

This appendix describes change you need to make to your Xcode and Visual C++ projects when upgrading from XOP Toolkit 5 to XOP Toolkit 6.

Xcode Project Changes For XOP Toolkit 6

This version of the manual describes Macintosh XOP development with Xcode 6 only. For details on using Xcode 3 or Xcode 4, download the XOP Toolkit 6.30 manual from:

<ftp://ftp.wavemetrics.net/IgorXOPToolkit/Manual/Old/XOPToolkitManual630.pdf>

In this section, we show the steps to convert the XFUNC1 project from XOP Toolkit 5 to XOP Toolkit 6 and Xcode 6.

Before proceeding with this section, you should familiarize yourself with the differences between XOP Toolkit 5 and XOP Toolkit 6. If you have not already done it, go to **XOP Toolkit 6 Upgrade Notes** on page 441 and familiarize yourself that material. Then return here.

It is assumed that you have copied your existing XOP Toolkit 5 project into your new IgorXOPs6 folder and that you are working on this copy. Substitute the name of your XOP for "XFUNC1".

You currently have a disk hierarchy that looks like this:

```
IgorXOP6
  XFUNC1
    Xcode
      <Xcode 2 or 3 project files here>
```

Here are the steps for converting to XOP Toolkit 6 and Xcode 6:

1-Xcode 6.

Rename Xcode folder in IgorXOPs6/XFUNC1 and as Xcode6.

2-Xcode 6.

Open the XFUNC1.xcodeproj file in Xcode 6.

3-Xcode 6.

Choose File→Project Settings and click the Build tab.

Click the Advanced button.

For Build Location, click the Legacy radio button.

Click Done to exit the Advanced dialog.

Click Done again to exit the Project Settings dialog.

4-Xcode 6.

Appendix B — Project Changes For XOP Toolkit 6

Choose View→Navigators→Show Project Navigator. The Project Navigator pane appears on the left.

Open all of the group icons in the Project Navigator so you can see all icons representing files.

4-Xcode 6.

Click the XFUNC1 icon at the top of the Project Navigator pane. The project settings appear in the center pane.

5-Xcode 6.

Choose View→Utilities→Show File Inspector.

In the File Inspector pane on the right, choose “Xcode 3.2-compatible” from the Project Format pop-up menu.

6-Xcode 6.

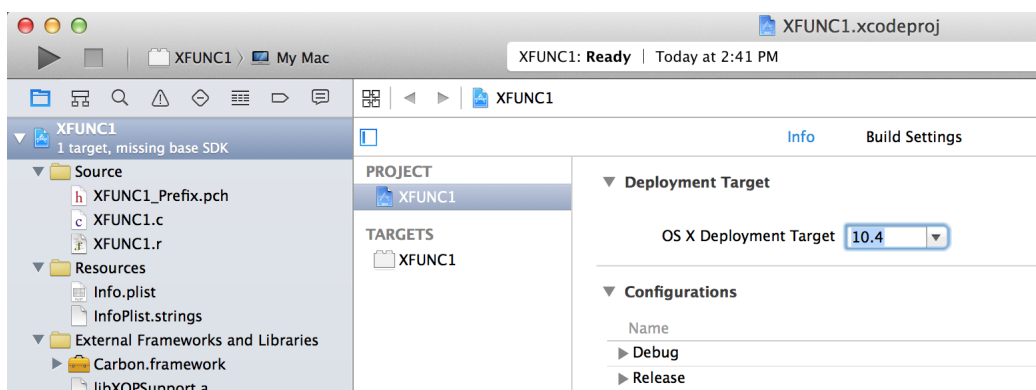
In the top/left corner of the center pane there is a small icon that looks like a rectangle with a vertical bar running down its left edge. Under it you should see the words PROJECT and TARGETS. If you don't see them, click the small icon to reveal the "Project and Targets List".

You should now see the word PROJECT with an XFUNC1 icon beneath it and TARGETS with another XFUNC1 icon beneath it in the left column of the center pane. This column is the Project and Targets List.

Click the XFUNC1 icon under the word PROJECT in the Project and Targets List.

Click the Info tab in the center pane.

You should see something like this:



Select 10.6 from the "OS X Deployment Target" popup menu. If 10.6 is not available, select the oldest OS version available that you want your XOP to run on.

7-Xcode 6.

Click the Build Settings tab.

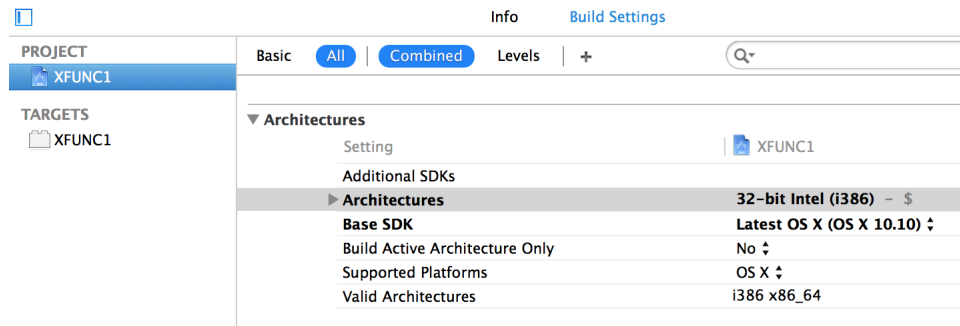
Click the All and Combined buttons.

Under Architectures/Architectures, select 32-bit Intel.

Under Architectures/Base SDK, select Latest OS X.

Under Build Options/Compiler for C/C++/Objective-C, select Default Compiler.

The project settings should look in part like this:



8-Xcode 6.

Click the XFUNC1 icon under the word TARGETS in the Project and Targets List.

Under Architectures/Architectures, select 32-bit Intel if it is not already selected.

Under Architectures/Base SDK, select Latest OS X if it is not already selected.

Under Build Options/Compiler for C/C++/Objective-C, select Default Compiler if it is not already selected.

Under Search Paths/Library Search Paths, enter "../XOPSupport/Xcode6" (without the quotes).

Under Apple LLVM Custom Compiler Flags, in the Other Warning Flags section, add "-Wno-parentheses" (without the quotes).

9-Xcode 6.

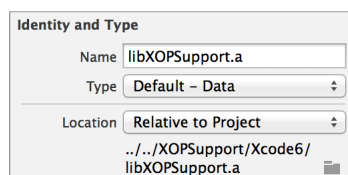
In the Project Navigator pane on the left, click the libXOPSupport.a icon

In the File Inspector pane on the right, select Relative to Project from the Location popup menu.

Click the folder icon under the right end of the Location popup menu to display the Choose File dialog.

Navigate to XOPSupport/Xcode6, select the libXOPSupport.a icon, and click the Choose button.

The File Inspector should look in part like this:



10-Xcode 6.

Click the XFUNC1 icon at the top of the Project Navigator pane.

Click the XFUNC1 icon under the word TARGETS in the Project and Targets List.

In the Rez section, under Other Rez Flags, delete "-d TARGET_RT_MACHO" if it is there.

11-Xcode 6.

Choose Product→Scheme→Edit Scheme. Select the XFUNC1 scheme from the popup menu at the top.

Click the Run icon in the list on the left.

Click the Info tab.

From the Executable popup menu, choose Other. In the resulting Open File dialog, navigate to your Igor Pro 32-bit application, select it, and click Choose.

Click Close.

12-Xcode 6.

Appendix B — Project Changes For XOP Toolkit 6

You most-likely will see a yellow issue icon near the top/center of the Xcode window, like this:



Click the issue icon to display the Issue Navigator in the lefthand pane.

You most-likely see a “Validate Project Settings” issue in the Issue Navigator. Click it.

Xcode displays a list of “Build Settings” to be changed. Most of these changes are either undesirable or of little consequence for XOP development.

Uncheck all of the checkboxes and click Done.

13-Xcode 6.

At this point your project has been updated for Xcode 6 but your source code still needs changes for XOP Toolkit 6.

Go to **Changes You Must Make** on page 442 and make the necessary source code changes. Then return here.

14-Xcode 6.

Choose Product→Build. The XOP should be successfully built.

To start the debugger choose Product→Run.

Visual C++ Project Changes For XOP Toolkit 6

This version of the manual describes Windows XOP development with Visual C++ 2015 only. For details on using older Visual C++ versions, download the XOP Toolkit 6.30 manual from:

<ftp://ftp.wavemetrics.net/IgorXOPToolkit/Manual/Old/XOPToolkitManual630.pdf>

Before proceeding with this section, you should familiarize yourself with the differences between XOP Toolkit 5 and XOP Toolkit 6. If you have not already done it, go to **XOP Toolkit 6 Upgrade Notes** on page 441 and familiarize yourself that material. Then return here.

In this section, we show the steps to convert the XFUNC1 Visual C++ solution and project files from VC 2005, also called VC 8 because the compiler version was 8, to VC 2015.

It is assumed that you have copied your existing XOP Toolkit 5 project into your new IgorXOPs6 folder and that you are working on this copy. Substitute the name of your XOP for “XFUNC1”.

You currently have a disk hierarchy that looks like this:

```
IgorXOP6
  XFUNC1
    VC8
      <VC8 project files here>
```

Here are the steps for converting to VC2015:

1-Visual C++ 2015.

Clone the VC8 folder in IgorXOPs6/XFUNC1 and rename the clone as VC2015.

2-Visual C++ 2015.

Launch VC2015.

3-Visual C++ 2015.

In VC2015, choose File→Open→Project/Solution.

4-Visual C++ 2015.

In the resulting Open File dialog, choose your IgorXOPs6/XFUNC1 /VC2015/XFUNC1.sln file and click Open.

Visual C++ will ask if you want to upgrade the solution.

5-Visual C++ 2015.

Tell Visual C+ to upgrade the solution.

Visual C++ opens a “migration report” in your web browser which lists issues that you need to address. We will address these issues in the following steps.

6-Visual C++ 2015.

In the Solution Explorer window, open the XFUNC1 icon and select the XOPSupport.lib icon.

Choose View→Other Windows→Properties Window.

In the Properties Window, in the Relative Path item, change VC8 to VC2015.

You are now linking with the VC2015 version of the XOPSupport library.

7-Visual C++ 2015.

Choose File→Save All.

8-Visual C++ 2015.

Quit Visual C++.

9-Visual C++ 2015.

On the desktop, delete the VC7 and VC8 folders in your IgorXOPs6/XFUNC1 folder.

Since XOP Toolkit 6 does not support VC7 and VC8, there is no need to keep the these project files.

10-Visual C++ 2015.

Delete everything from the IgorXOPs6/XFUNC1/VC2015 folder except for these files:

XFUNC1.sln

XFUNC1.vcxproj

XFUNC1.vcxproj.filters

Note that the .vcproj file is to be deleted as it is superceded by the .vcxproj file.

Modifying Your VC2015 Project For XOP Toolkit 6

In this section, we modify the VC2015 XFUNC1 project created in the preceding section to use settings appropriate for XOP Toolkit 6.

It is assumed that you have copied your existing XOP Toolkit 5 project into your new IgorXOPs6 folder and that you are working on this copy. Your original project will remain unmodified in your IgorXOPs5 folder.

The main changes relative to XOP Toolkit 5 are:

- The recommended structure alignment is Default instead of 2 Bytes
- The recommended values for some settings have been updated to reflect VC2015 practices

An example of the latter is using "\$(Configuration)\\" instead of "\$(ConfigurationName)" or "Debug" and "Release" for the Intermediate Directory setting. The new way does the same thing using the VC2015 technique.

Also, some settings (e.g., Target Extension) that are set incorrectly by the VC2015 conversion routine are corrected in the following steps.

You will need to substitute the name of your XOP for "XFUNC1".

Appendix B — Project Changes For XOP Toolkit 6

Explicit instructions are given for the important settings which need to be changed from the default Visual C++ settings. Screen dumps then show you what your setting should look like. You may see differences between your settings and the screen dumps. Generally these differences are in settings that are not critical for XOP development. If you understand what they are for, you may wish to leave them as they are. Otherwise, change them to match the screen dump.

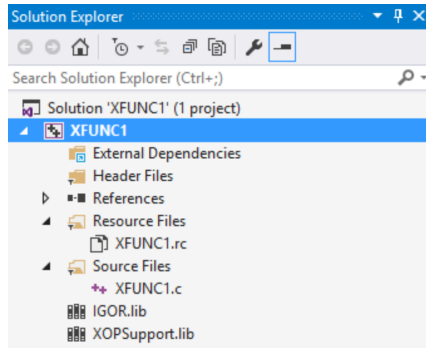
1-Visual C++ 2015.

Open the XFUNC1\VC2015\XFUNC1.sln solution file in VC 2015.

If you have multiple versions of Visual C++ on your system, the wrong version might start up. If that happens, cancel and quit the wrong version. Start the right version and use File→Open Project to open XFUNC1.sln.

2-Visual C++ 2015.

In the Solution Explorer window, select the XFUNC1 project icon (not the solution icon).



3-Visual C++ 2015.

Choose Project→Properties.

The XFUNC1 Property Pages window opens.

Appendix B — Project Changes For XOP Toolkit 6

4-Visual C++ 2015.

Open the Configuration Properties node of the tree view control.

Select General from the Configuration Properties node.

Select All Configurations from the Configuration popup menu.

Enter the following settings:

General

Output Directory: .\

Intermediate Directory: \$(Configuration)\

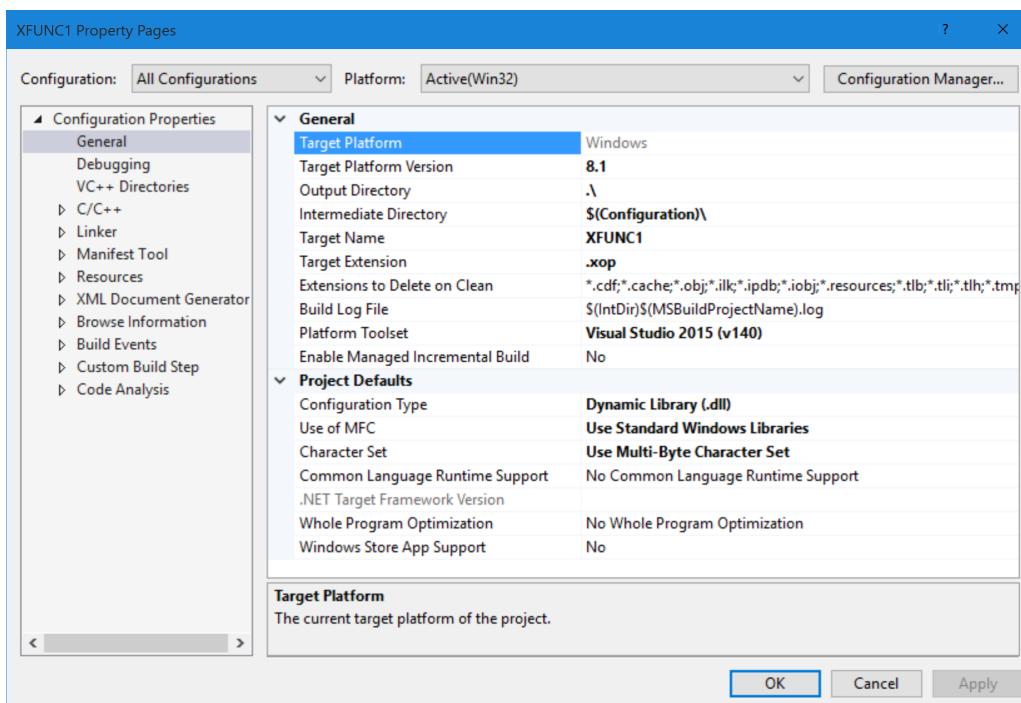
Target Name: XFUNC1

Target Extension: .xop

Character Set: Use Multi-Byte Character Set

Click the **Apply** button.

The Property Pages window now looks like this:



5-Visual C++ 2015.

Select All Configurations from the Configuration popup menu.

Select Debugging from the Configuration Properties node of the tree view control.

Enter the following setting:

Debugging

Command: <path to your 32-bit Igor.exe>

where <path to your Igor.exe> is typically something like

C:\Program Files (x86)\WaveMetrics\Igor Pro Folder\Igor.exe

You can set it using the Browse item in the pop-down menu for the Command item.

Click the Apply button.

6-Visual C++ 2015.

Open the C/C++ node in the tree view control.

7-Visual C++ 2015.

Select General from the C/C++ node of the tree view control.

Select All Configurations from the Configuration popup menu.

Enter the following setting:

C/C++

General

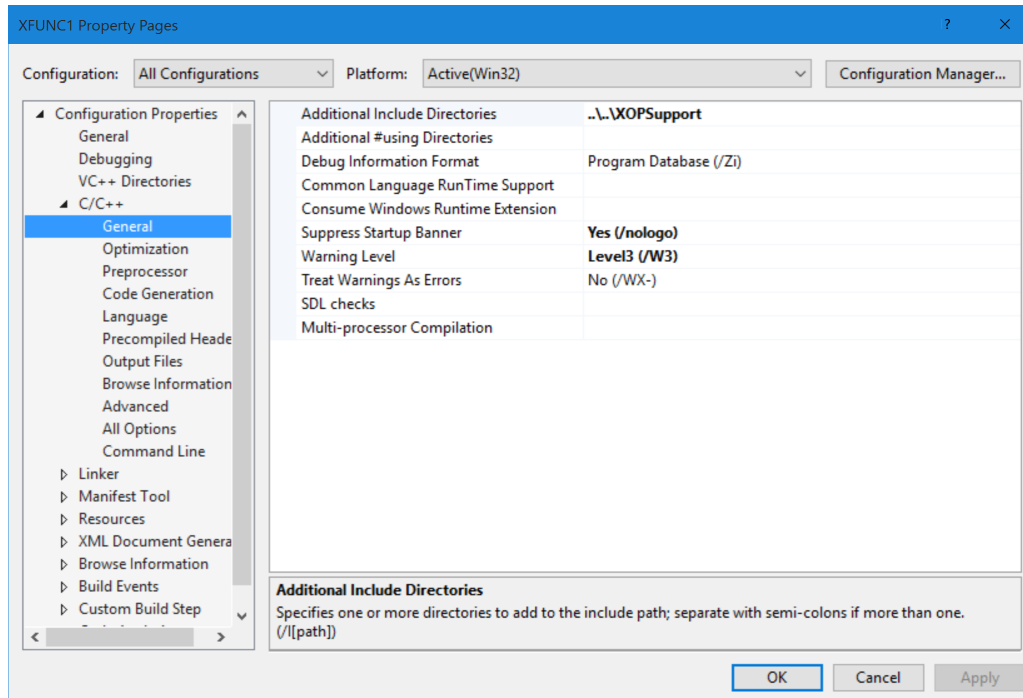
Additional Include Directories: ..\..\XOPSupport

Debug Information Format: Program Database (/Zi)

Click the Apply button.

Appendix B — Project Changes For XOP Toolkit 6

The Property Pages window now looks like this:



8-Visual C++ 2015.

Select Debug from the Configuration popup menu.

Select Preprocessor from the C/C++ node of the tree view control.

Enter the following in the Preprocessor Definitions setting:

WIN32;_DEBUG;_WINDOWS;_USRDLL;_CRT_SECURE_NO_WARNINGS

The older _CRT_SECURE_NO_DEPRECATED macro also works but Microsoft now recommends _CRT_SECURE_NO_WARNINGS.

VC2015 automatically adds “%(PreprocessorDefinitions)”. This is required.

Click the Apply button.

Select Release from the Configuration popup menu.

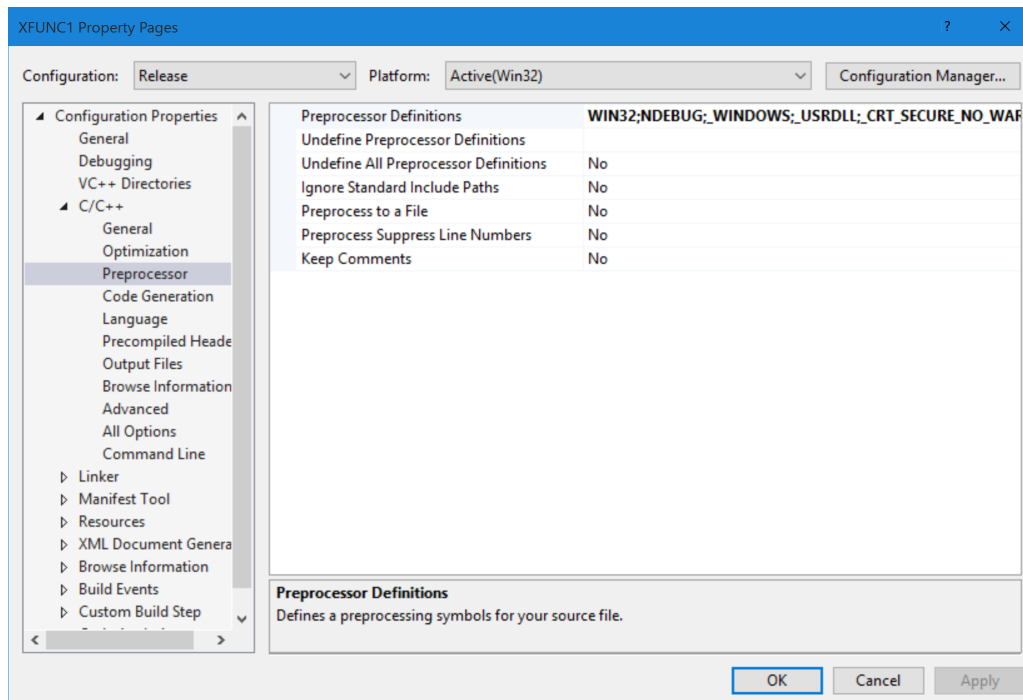
Enter the following in the Preprocessor Definitions setting:

WIN32;NDEBUG;_WINDOWS;_USRDLL;_CRT_SECURE_NO_WARNINGS

VC2015 automatically adds “%(PreprocessorDefinitions)”. This is required. VC2015 adds it automatically, similarly to the Additional Include Directories setting described above.

Click the Apply button.

The Property Pages window now looks like this:



Appendix B — Project Changes For XOP Toolkit 6

9-Visual C++ 2015.

Select Debug from the Configuration popup menu.

Select Code Generation from the C/C++ node of the tree view control.

Enter the following settings:

C/C++

Code Generation

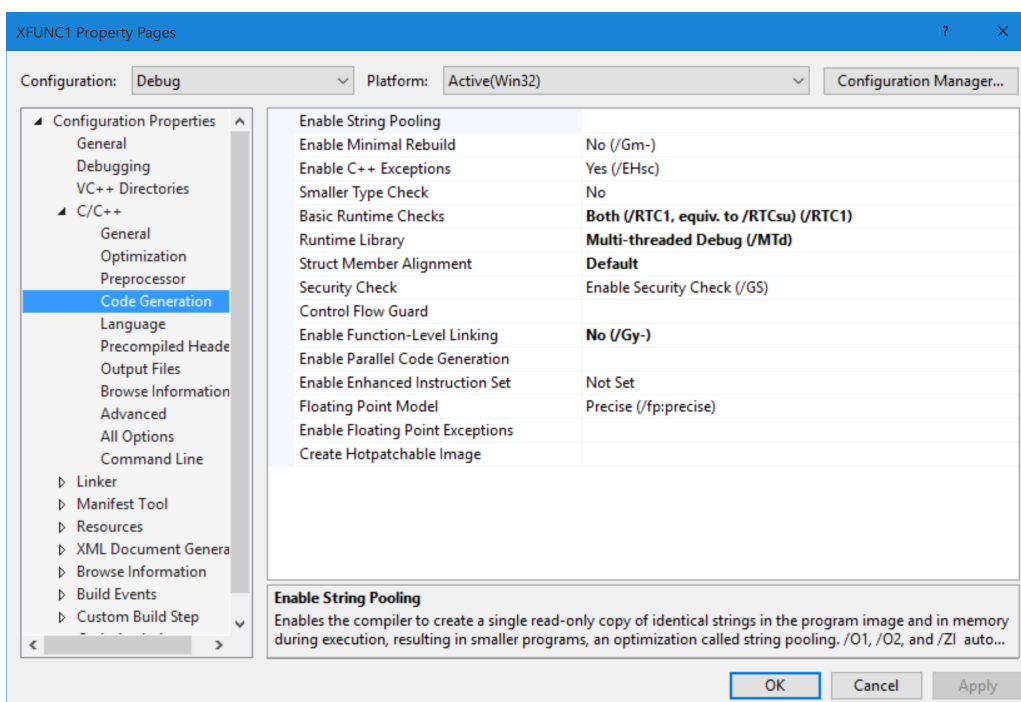
Runtime Library: Multi-threaded Debug (/MTd)

Struct Member Alignment: Default

Enable Function Level Linking: No

Click the Apply button.

The Property Pages window now looks like this:



10-Visual C++ 2015.

Select Release from the Configuration popup menu.

Select Code Generation from the C/C++ node of the tree view control.

Enter the following settings:

C/C++

Code Generation

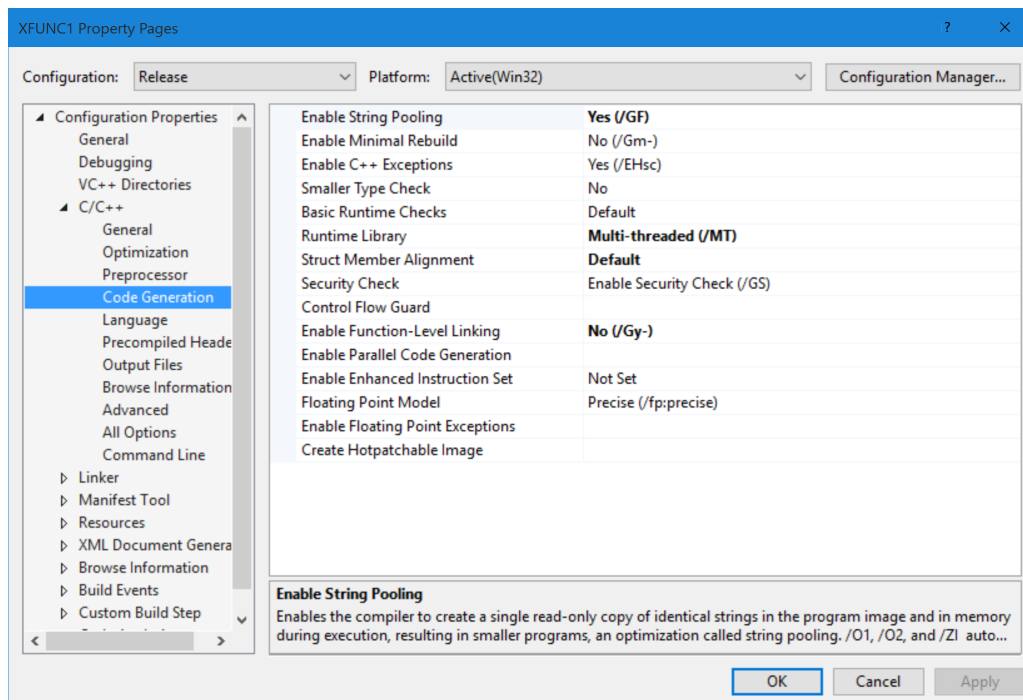
Runtime Library: Multi-threaded (/MT)

Struct Member Alignment: Default

Enable Function Level Linking: No

Click the Apply button.

The Property Pages window now looks like this:



Appendix B — Project Changes For XOP Toolkit 6

11-Visual C++ 2015.

Select All Configurations from the Configuration popup menu.

Select Precompiled Headers from the C/C++ node of the tree view control.

Enter the following settings:

C/C++

Precompiled Headers

Create/Use Precompiled Header: Not Using Precompiled Headers

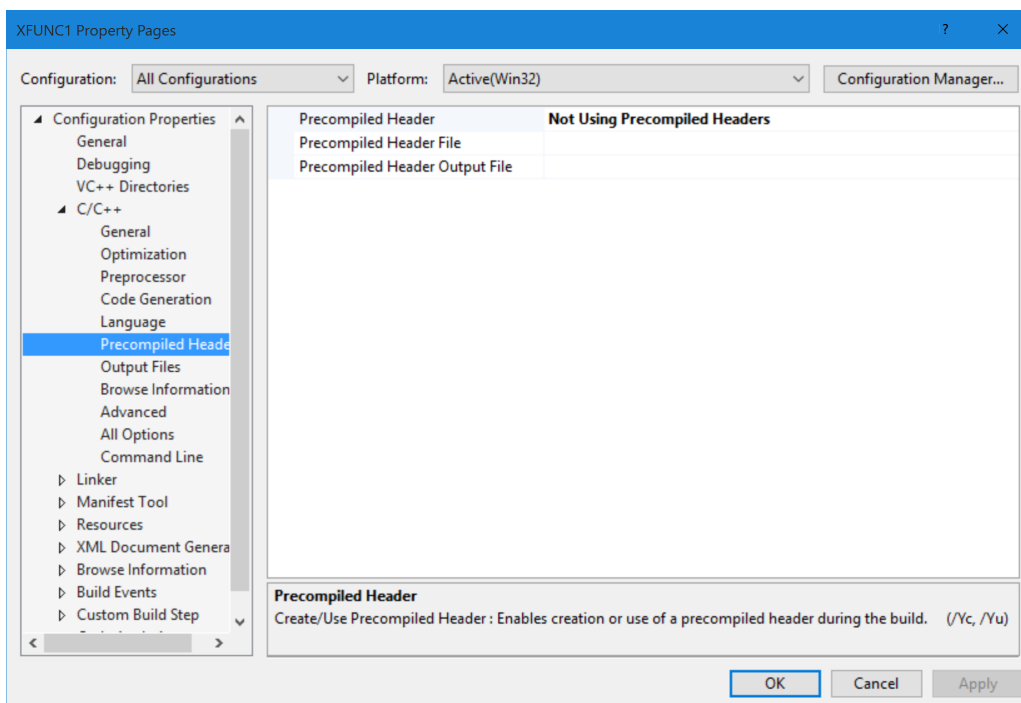
Create/Use PCH Through File:

Precompiled Header Output File:

Advanced programmers can set up precompiled headers but it is usually not worth the effort since XOPs tend to have a small number of source files.

Click the Apply button.

The Property Pages window now looks like this:



12-Visual C++ 2015.

Select All Configurations from the Configuration popup menu.

Select Output Files from the C/C++ node of the tree view control.

Enter the following settings:

C/C++

Output Files

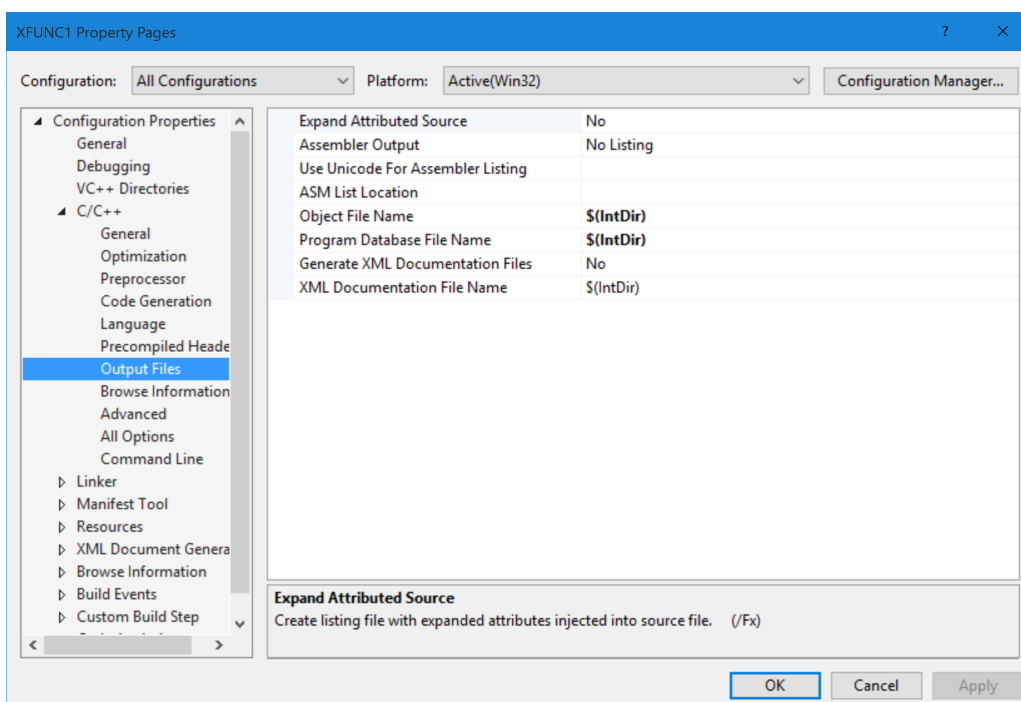
Object File Name: \$(IntDir)

Program Database File Name: \$(IntDir)

Note that there must be no backslash characters in these settings.

Click the **Apply** button.

The Property Pages window now looks like this:



Appendix B — Project Changes For XOP Toolkit 6

13-Visual C++ 2015.

Select All Configurations from the Configuration popup menu.

Select Browse Information from the C/C++ node of the tree view control.

Enter the following settings:

C/C++

Browse Information

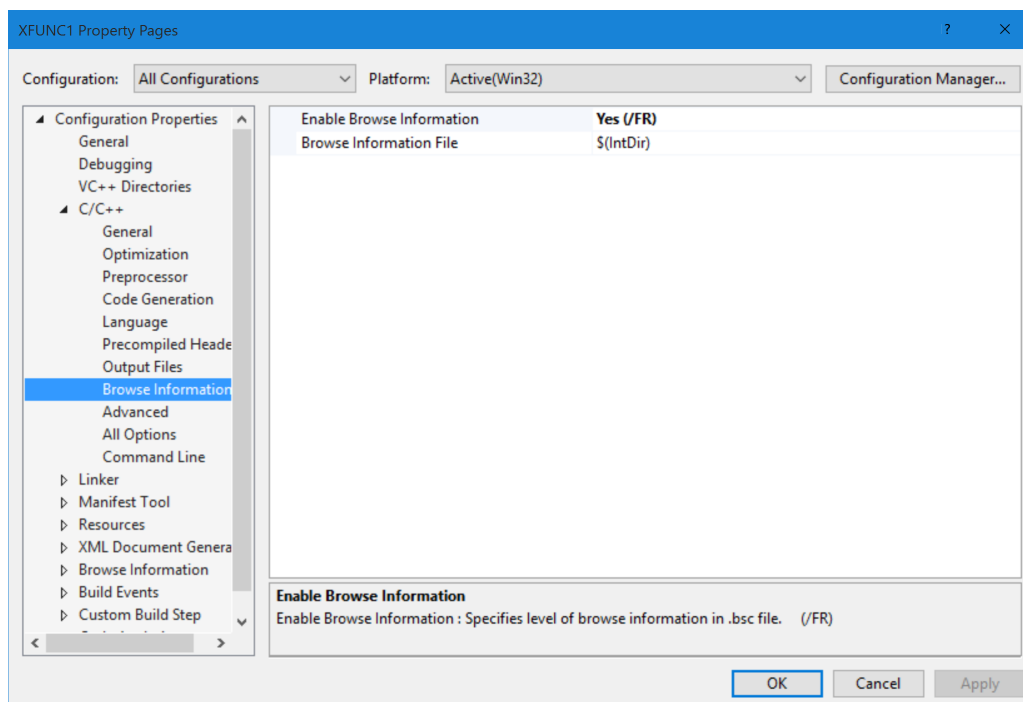
Enable Browse Information: Yes (/FR)

Browse File: \$(IntDir)

Note that there must be no backslash character in the Browse File setting.

Click the **Apply** button.

The Property Pages window now looks like this:



14-Visual C++ 2015.

Select All Configurations from the Configuration popup menu.

Select Advanced from the C/C++ node of the tree view control.

Enter the following setting:

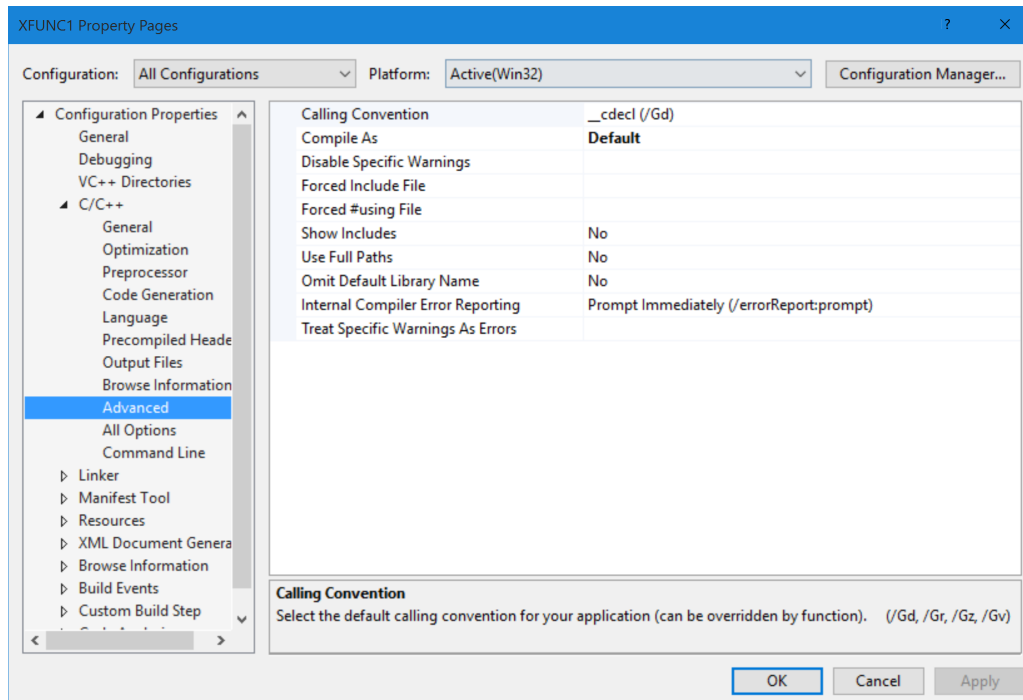
C/C++

Advanced

Compile As: Default

Click the Apply button.

The Property Pages window now looks like this:



15-Visual C++ 2015.

Select All Configurations from the Configuration popup menu.

Select Command Line from the C/C++ node of the tree view control.

Delete anything in the Additional Options section.

Click the Apply button.

16-Visual C++ 2015.

Click OK.

The Property Pages window closes.

17-Visual C++ 2015.

Choose File→Save All.

Your C++ settings are now saved. Now we will move to the linker settings.

18-Visual C++ 2015.

Right-click the XFUNC1 project icon in the Solution Explorer and choose Properties.

In the tree view control, open the Configuration Properties and Linker nodes.

19-Visual C++ 2015.

Select **General** from the **Linker** node of the tree view control.

Select **All Configurations** from the **Configuration** popup menu.

Enter the following setting:

Linker

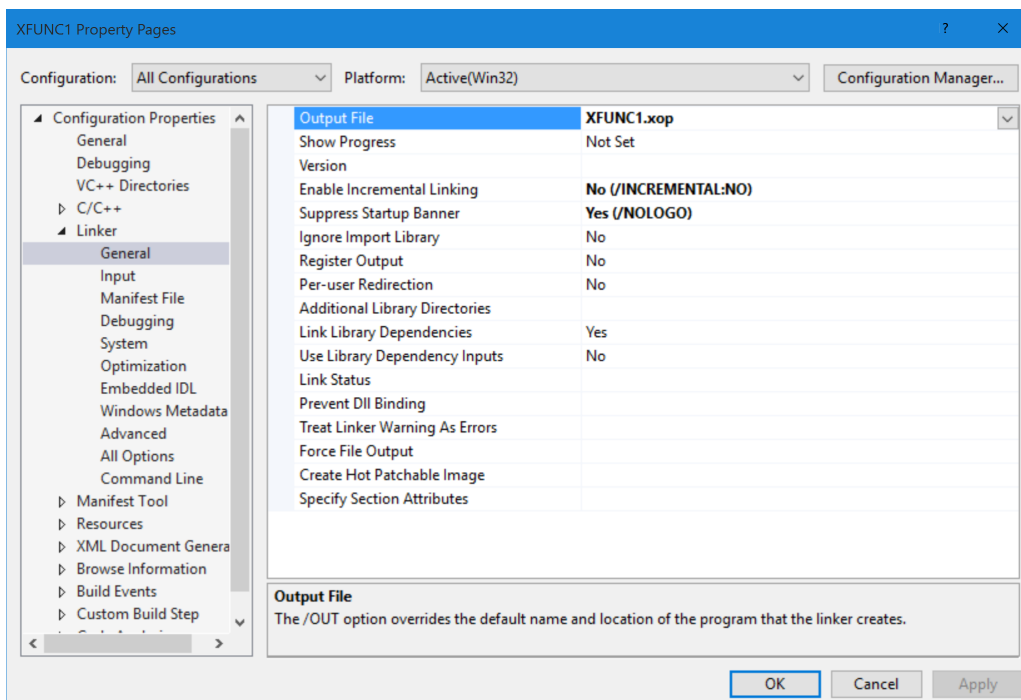
General

Output File: XFUNC1.xop

Incremental Linking: No

Click the Apply button.

The Property Pages window now looks like this:



Not shown is the fact that incremental linking is turned on for the Debug configuration and off for the Release configuration. These are the Visual C++ defaults for incremental linking.

Appendix B — Project Changes For XOP Toolkit 6

20-Visual C++ 2015.

Select All Configurations from the Configuration popup menu.

Select Input from the Linker node of the tree view control.

Enter the following setting:

Linker

Input

Additional Dependencies: version.lib

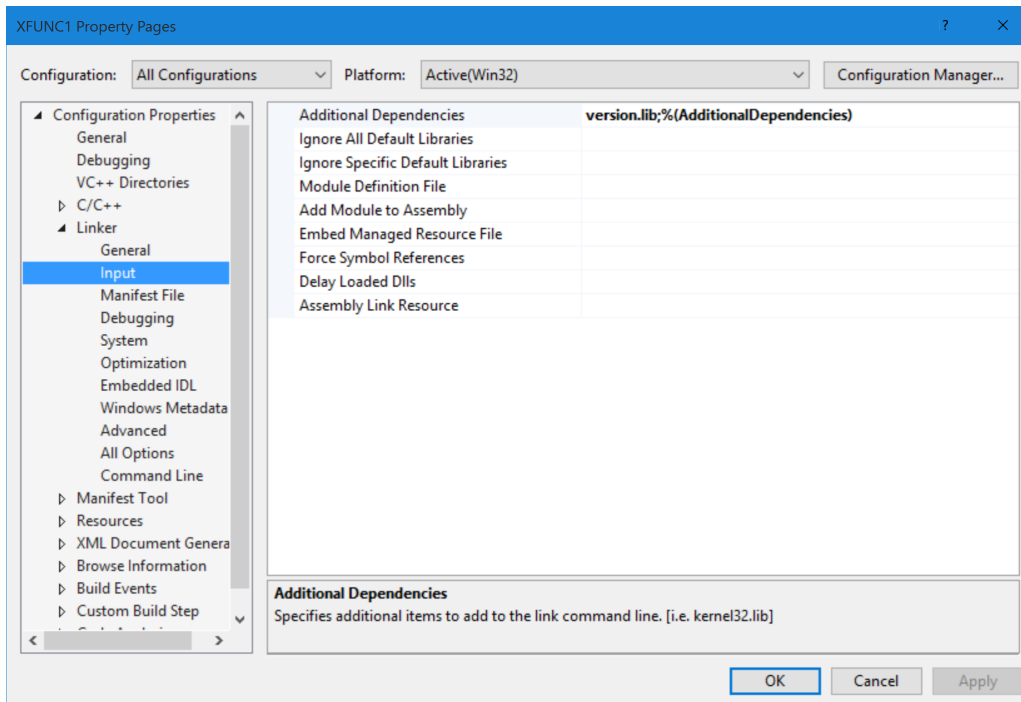
Ignore All Default Libraries:

VC2015 automatically adds “%(AdditionalDependencies)” to the Additional Dependencies setting. This is required.

The Ignore All Default Libraries setting can be blank although it is not critical.

Click the Apply button.

The Property Pages window now looks like this:



21-Visual C++ 2015.

Select All Configurations from the Configuration popup menu.

Select Debugging from the Linker node of the tree view control.

Enter the following setting:

Linker

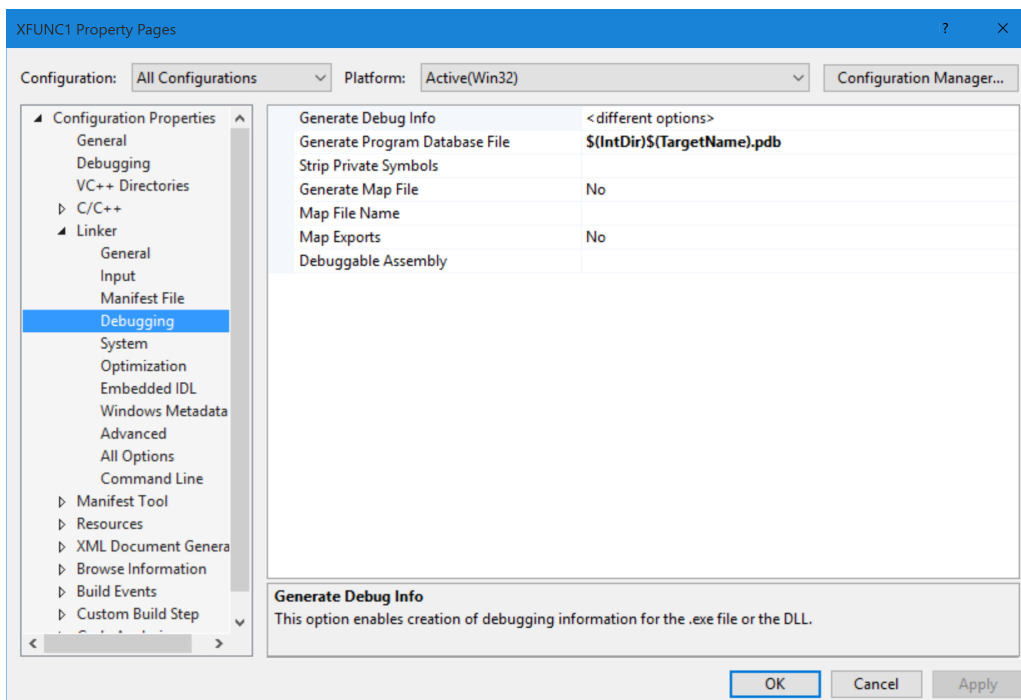
Debugging

Generate Program Database File: \$(IntDir)\$(TargetName).pdb

Note that there must be no backslash character in this setting.

Click the Apply button.

The Property Pages window now looks like this:



Appendix B — Project Changes For XOP Toolkit 6

22-Visual C++ 2015.

Select All Configurations from the Configuration popup menu.

Select System from the Linker node of the tree view control.

Enter the following settings:

Linker

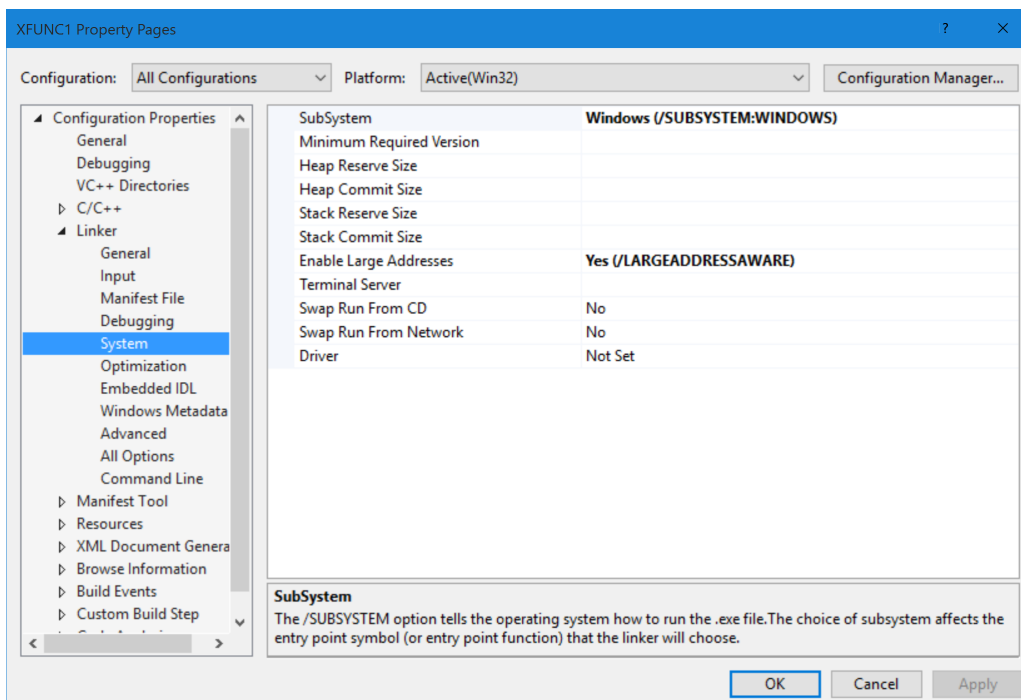
System

SubSystem: Windows (/SUBSYSTEM:WINDOWS)

Enable Large Addresses: Yes (/LARGEADDRESSAWARE)

Click the Apply button.

The Property Pages window now looks like this:



23-Visual C++ 2015.

Select All Configurations from the Configuration popup menu.

Select Advanced from the Linker node of the tree view control.

Enter the following settings:

Linker

Advanced

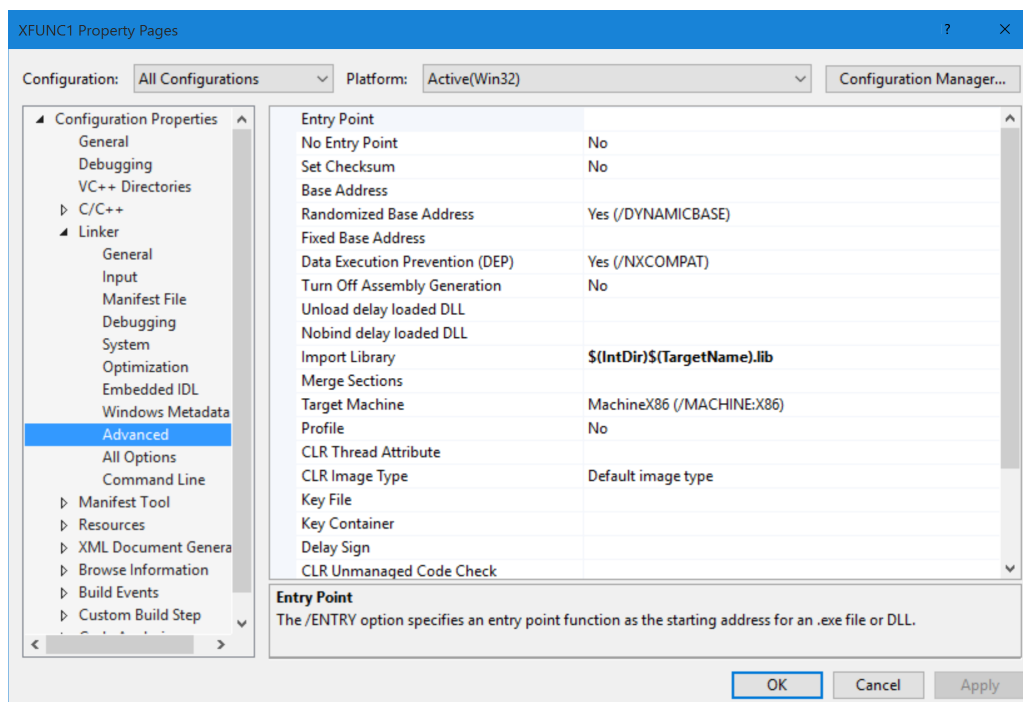
Import Library: \$(IntDir)\$(TargetName).lib

Target Machine: MachineX86 (/MACHINE:X86)

Note that there must be no backslash character in the Import Library setting.

Click the Apply button.

The Property Pages window now looks like this:



24-Visual C++ 2015.

Select All Configurations from the Configuration popup menu.

Select Command Line from the Linker node of the tree view control.

Delete anything in the Additional Options section.

Old projects may contain /MACHINE:I386 here. This is not needed and will cause problems if you later add a 64-bit platform to your project.

Click the Apply button.

25-Visual C++ 2015.

Click OK.

The Property Pages window closes.

26-Visual C++ 2015.

Choose File→Save All.

Your Linker settings are now saved. Now we will move to the resource settings.

27-Visual C++ 2015.

Right-click the XFUNC1 project icon in the Solution Explorer and choose Properties.

In the tree view control, open the Configuration Properties and Resources nodes.

28-Visual C++ 2015.

Select General from the Resources node of the tree view control.

Select All Configurations from the Configuration popup menu.

Enter the following setting:

Resources

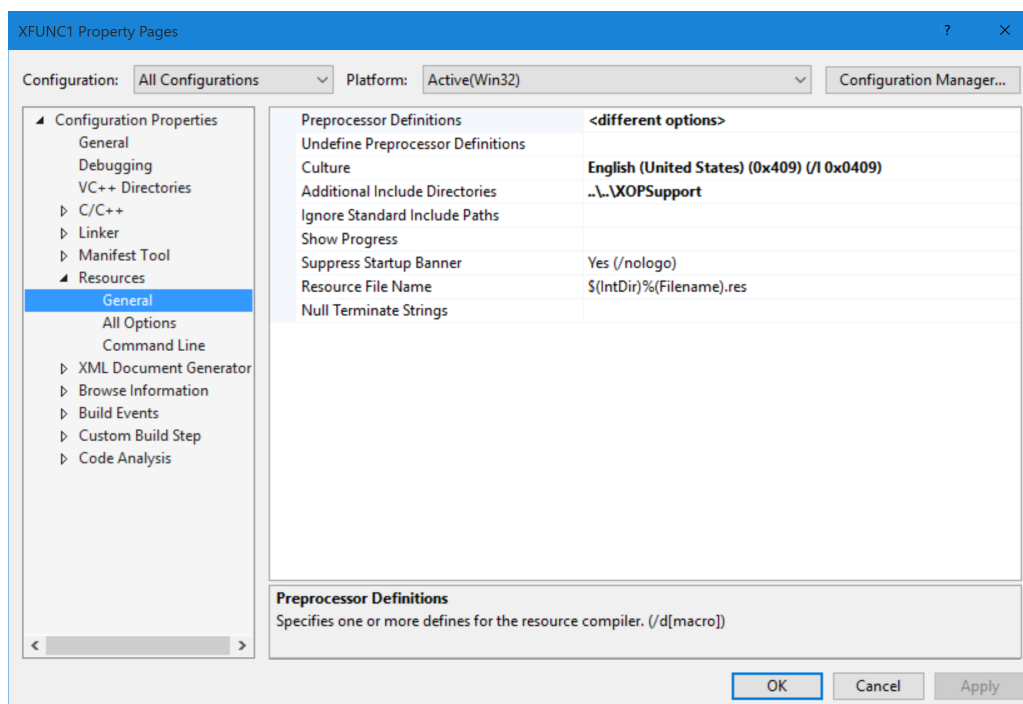
General

Additional Include Directories: ..\..\XOPSupport

VC2015 automatically adds “%(AdditionalIncludeDirectories)”. This is not needed and can be left or removed.

Click the Apply button.

The Property Pages window now looks like this:



29-Visual C++ 2015.

Click OK.

The Property Pages window closes.

30-Visual C++ 2015.

Choose File→Save All.

Your project settings are now set up for XOP Toolkit 6 development.

31-Visual C++ 2015.

In this step we make sure that your XOPSupport library is correct. If you followed the steps under **Modifying Your VC2015 Project For XOP Toolkit 6** on page 461 then you have already done this and don't need to repeat it.

In the Solution Explorer window, right-click the XOPSupport.lib icon and choose Remove.

In the Solution Explorer window, right-click the XFUNC1 project icon and choose Add→Existing Item.

An Open File dialog is displayed.

Navigate to the IgorXOPs6\XOPSupport\VC2015 directory.

Select All Files from the popup menu to the right of the File Name edit box.

Select the XOPSupport.lib file and click Add.

If you get a dialog asking about "custom build rules", click No.

32-Visual C++ 2015.

Choose File→Save All.

Your VC2015 project is now set up for XOP Toolkit 6 development.

XOP Toolkit 6 Release History

Overview

This appendix describes changes in each new release of XOP Toolkit 6.

For a discussion of what is new in XOP Toolkit 6 relative to XOP Toolkit 5, see **XOP Toolkit 6 Upgrade Notes** on page 441.

This manual documents XOP Toolkit 6.40. XOP Toolkit 6.40 creates XOPs that require Igor Pro 6.20 or later.

Release 6.00

October 25, 2010

Initial release.

Release 6.01

June 16, 2011

Fixed the FunctionInfo structure to include the isThreadSafe field which was inadvertently omitted in release 6.00.

Added sample projects and documentation for XOP development using Xcode 4.

Changed the Thread Safety sections in the reference chapter to make it clear when a callback is Igor-thread-safe but not safe to call from a private internal thread created by the XOP.

Added a section on multi-threaded data acquisition. See **Strategies For Threaded Data Acquisition** on page 174.

Added the **ThreadGroupPutDF** and **ThreadGroupGetDF** callbacks. These require Igor Pro 6.23 or later.

Release 6.02

August 22, 2011

Updated the documentation and Xcode 4 projects for Xcode 4.1.

Release 6.30

February 27, 2013

This release is mostly to keep up with Xcode 4 changes and to add Visual C++ 2012 sample XOPs and documentation. As described below, a side-effect of keeping up with Xcode 4 is that XOPs compiled by XOP

Appendix C — XOP Toolkit 6 Release History

Toolkit 6.30 require Igor Pro 6.20 or later. The requirement of Igor Pro 6.20 is the reason for bumping the XOP Toolkit version from 6.02 to 6.30. 6.30 was chosen because that is the current Igor version. The following paragraphs describe the changes in more detail.

Dropped support for Xcode versions prior to 3.2.

Updated the documentation and Xcode 4 projects for Xcode 4.3.2 or later.

Added Visual C++ 2012 sample projects and documentation.

Changed all Xcode 4 projects to use the LLVM compiler because Apple is dropping support for GCC. This required converting all sample XOP projects to use XOPMain instead of main as the main function. XOPs that use XOPMain require Igor Pro 6.20 or later. For further information see **The XOPMain Function** on page 215.

Added file XOPSupport/VC10/vc90.pdb file to the distribution. This prevents VC10 and VC2012 from emitting warning LNK4099 ("vc90.pdb not found") when you compile the debug configuration of a 32-bit XOP. You will still get the warning when you compile a 64-bit XOP because the vc90.pdb file that was added is for the 32-bit XOPSupport library.

Changed all ParseOperationTemplate /C=2 calls to use /C=6. In Igor Pro 6.22 or later, /C=6 generates extern "C" in the declaration of the ExecuteOperation function. See **C++ Code Called From C Code** on page 218 for details.

The 'vs' keyword is allowed in Operation Handler operations. See **The Command Template** on page 74 for details. This requires Igor Pro 6.30.

Operation Handler now supports "qualified keywords" like this:

```
ModifyGraph mode(wave0) = 3    // wave0 is the qualifier for the mode keyword
```

For details see **Qualified Keywords** on page 80.

Added const qualifiers to the Windows declarations of parameters for several XOPSupport routines: setmenutext, insertmenuitem, appendmenu, PtrToHand, and PtrAndHand and changed PtrToHand to take a const void* instead of a Ptr. This made the Windows declarations consistent with the corresponding Macintosh declarations. The change to PtrAndHand fixed a Windows compile error in WaveAccess.c that was introduced in XOP Toolkit 6.01.

Added a const qualifier to a parameter of SetScrapData.

Fixed a bug in PositionWinDialogWindow that caused the dialog to appear partially offscreen when working with multiple monitors under some circumstances if Igor was running on the second monitor.

Added a note to the documentation for **CheckAbort** explaining that it may not work correctly with non-English keyboards.

The documentation for **Wave Reference Counting** on page 128 and **Free Waves** on page 132 was revised and extended.

In Igor Pro 6.30, the **SetTextWaveData** routine was changed to return an error if the wave is locked or if the wave was passed to a thread as a parameter to the thread worker function. This change is in Igor, not in the XOP Toolkit.

Change To HoldWave

Removed the second parameter from the **HoldWave** XOPSupport routine. Previously the function was defined as:

```
int HoldWave(waveHndl waveH, waveHndl* waveRefPtr)
```

This proved to be confusing. So, as of XOP Toolkit 6.30, the function is defined as:

```
int HoldWave(waveHndl waveH)
```

This change does not affect XOPs compiled with earlier versions of the XOP Toolkit. It will cause a compile error that you will have to correct if you have used HoldWave and compile with XOP Toolkit 6.30 or later.

For further information, see **Wave Reference Counting** on page 128.

Change To HoldDataFolder

Removed the second parameter from the **HoldDataFolder** XOPSupport routine. Previously the function was defined as:

```
int HoldDataFolder(DataFolderHandle dfH, DataFolderHandle* dfRefPtr)
```

This proved to be confusing. So, as of XOP Toolkit 6.30, the function is defined as:

```
int HoldDataFolder(DataFolderHandle dfH)
```

This change does not affect XOPs compiled with earlier versions of the XOP Toolkit. It will cause a compile error that you will have to correct if you have used HoldDataFolder and compile with XOP Toolkit 6.30 or later.

For further information, see **Data Folder Reference Counting** on page 140.

Release 6.40

October 16, 2015

XOP Toolkit 6.40 requires Igor Pro 6.20 or later. The main reason for this is the restriction imposed by modern compilers as explained under **The XOPMain Function** on page 215. Secondary reasons include changes to support multi-threading, wave references and data folder references.

Make sure that your main function is named XOPMain as explained under **The XOPMain Function** on page 215. Make sure that it checks the version of Igor it is running with. See **Checking Igor's Version** on page 66 for details.

On Macintosh, changed the Xcode XOPSupport project to be compiled using the LLVM 5.1 compiler instead of the LLVM-GCC4.2 compiler which Apple no longer supports.

Changed SINGLE_NAN and SetNaN32 on Windows to return a quiet NaN represented as 0x7FC00000. This is what is returned on Macintosh and by the numeric_limits<float>::quiet_NaN() function. Previously they returned 0xFFFFFFFF.

Changed DOUBLE_NAN and SetNaN64 on Windows to return a quiet NaN represented as 0x7FF8000000000000. This is what is returned on Macintosh and by the numeric_limits<double>::quiet_NaN() function. Previously they returned 0x7FFFFFFFFFFFFFFF.

Added a space between the quotation mark and CR_STR as required by VC2015. See "String literals followed by macros" at <https://msdn.microsoft.com/en-us/library/Bb531344.aspx>. For example, this:

```
XOPNotice("Test"CR_STR);
```

must be changed to this:

```
XOPNotice("Test" CR_STR);
```

to prevent errors when compiling with VC2015.

Added the **WaveMemorySize** callback. It works with Igor Pro 3 or later. This was added for the Data Browser XOP and is of no use for most XOPs.

Added the GET_IGOR_VERSION XOPSupport callback code. This is used by GetIgorVersion if you are running with Igor7 or later. You don't need to use it directly.

Appendix C — XOP Toolkit 6 Release History

Fixed an obscure bug in SanitizeWaveName.

Changes To XOP Menu Support

A new set of XOPSupport routines was added for dealing with menus. You need to know this only if your XOP adds, removes, or modifies menus or menu items.

The new menu routines are designed for future compatibility with Igor Pro 7 but they work with any version of Igor. They are described under **XOPSupport Menu Routines** on page 143.

The old menu routines are no longer supported. New code must use the new menu routines and existing code must be updated to use the new routines. See the section “XOP Menu Compatibility” in XOPMenus.c for details.

The sample XOPs GBLoadWave, MenuXOP1, TUDemo, VDT2, and WindowXOP1 were changed to use the new menu routines.

This table shows the correspondence between old and new routines:

Old Menu Routine	New Menu Routine
GetMenuHandle	XOPResourceMenuIDToMenuRef
ResourceMenuIDToMenuHandle	XOPResourceMenuIDToMenuRef
WMGetMenuFromModule	None
GetMenuID	XOPGetMenuInfo
SetMenuID	None
CountMItems	XOPCountMenuItems
WMDrawMenuBar	None
WMInsertMenu	XOPShowMainMenu
WMDeleteMenu	XOPHideMainMenu
None	XOPGetMenuItemInfo
getmenuItemtext	XOPGetMenuItemText
setmenuItemtext	XOPSetMenuItemText
appendmenu	XOPAppendMenuItem
insertmenuItem	XOPInsertMenuItem
DeleteMenuItem	XOPDeleteMenuItem
WMDeleteMenuItems	XOPDeleteMenuItemRange
EnableItem	XOPEnableMenuItem
DisableItem	XOPDisableMenuItem
CheckItem	XOPCheckMenuItem
FillMenu	XOPFillMenu
FillMenuNoMeta	XOPFillMenuNoMeta
FillWaveMenu	XOPFillWaveMenu
FillPathMenu	XOPFillPathMenu
FillWinMenu	XOPFillWinMenu

Xcode6 Projects

XOP Toolkit 6.40 adds Xcode6 projects. The projects were tested with Xcode 6.1, Xcode 6.4 and Xcode 7.01. They are configured to require OS X 10.6 or later.

The Xcode6 XOPSupport library does not support XOP dialog routines because support for the old Macintosh Dialog Manager was removed from recent Apple SDK's. If your XOP requires XOP dialog support, you must use the Xcode4 version of the project and you must continue to use an old Apple SDK such as the 10.6 SDK. Alternatively, use XOP Toolkit 7 which implements dialogs using the Cocoa API instead of the Carbon API.

Because the GBLoadWave and VDT2 XOPs implement dialogs using XOP dialog support routines, there are no Xcode6 projects for these XOPs.

The Xcode6 XOPSupport library also does not support XOP windows because support for the old Macintosh Window Manager was removed from recent Apple SDK's. There is no Xcode6 project for the WindowXOP1 project because it uses antiquated Macintosh techniques.

If you update your project to Xcode6, change it to point to the Xcode6 version of the XOPSupport library. Otherwise you may get strange link errors.

Visual C++ 2015 Projects

XOP Toolkit 6.40 adds Visual C++ 2015 projects.

If you update your project to VC2015, change it to point to the VC2015 version of the XOPSupport library. Otherwise you may get strange link errors.

To get rid of annoying "type cast conversion from 'a' to 'b' of greater size" warnings, we disabled warning 4312 in the VC2015 sample XOP projects. You should do the same for your project, as follows:

- 1 Open the project properties window.**
- 2 From the Configuration popup menu, choose All Configurations.**
- 3 From the Platform popup menu, choose All Platforms.**
- 4 In the C/C++ section, select the Advanced subsection.**
- 5 Under Disable Specific Warnings, enter "4312;" without the quotes.**
- 6 Click OK.**

VC2015 does not permit constructions like this:

```
strcpy(buffer, "Literal string"CR_STR);
```

This change is explained under "String literals followed by macros" at <https://msdn.microsoft.com/en-us/library/Bb531344.aspx>.

To cope with this, we had to add a space before CR_STR to all such occurrences, like this:

```
strcpy(buffer, "Literal string" CR_STR);
```

You should do the same in your project.

Symbols

- #defines
 - _MACINTOSH_ 442
 - _WINDOWS_ 442
 - XOP Toolkit 6 443
- Wno-parentheses 38
- _CRT_SECURE_NO_WARNINGS 45
- _MACINTOSH_ #define
 - replaced by MACIGOR 442
- _WIN32_WINNT 450
- _WINDOWS_ #define
 - replaced by WINIGOR 442
- .c files 8
- .DS_Store file 44
- .exp files
 - Xcode 32, 38
- .ihf extension 219
- .pdb files 46
- .r files 8
 - compiled by Rez 52
- .rc files 8, 52
 - Visual C++ 44
- .rsrc files
 - in packages 52
- .sln files 44
- .vcxproj files 44
- .xop extension 11, 49

Numbers

- 64-bit XOPs 181, 453
 - changed parameter types 453
 - command help 195
 - data types 190–191
 - Igor Extensions (64-bits) 181
 - Igor Pro User Files folder 181
 - long data type 190, 192
 - Macintosh 181
 - porting to 64-bits 182
 - printf and sprintf issues 195
 - programming issues 191
 - virtual memory 181
 - Visual C++ 182
 - Windows 181

A

- aborting 216
 - CheckAbort 420
- About Igor dialog 16
- accessing data folders
 - XOPSupport routines 298–313

- accessing variables
 - XOPSupport routines 289–297
- accessing waves
 - XOPSupport routines 257–288
- ACTIVATE message 157
 - not sent on Windows 155
- activating
 - windows 164
- activating XOPs 5
- Activity Monitor
 - leaks 231
- ActualToResourceItem 150, 315
- ActualToResourceMenuID 149, 314
- adding commands 69–96
- adding dialogs 203–209
- adding functions 97–117
- adding menus and menu items 143–152
- adding operations 69–96
- adding windows 153–165
- AddPopupMenuItems 334
- afxres.h 451
- alerts
 - custom error alert 60
 - XOPOKAlert 433
 - XOPOKCancelAlert 433
 - XOPYesNoAlert 433
 - XOPYesNoCancelAlert 433
- aliases 11
- appendmenu 486
- ArrowCursor 362
- as keyword 74
 - optional parameters 76
- Asian text
 - ConcatenatePaths 340
 - FullPathPointsToFile 342
 - FullPathPointsToFolder 342
 - GetDirectoryAndFileNameFromFullPath 341
 - GetNativePath 340
 - HFSToPosixPath 339
 - MacToWinPath 338
 - strchr2 428
 - strrchr2 428
 - WinToMacPath 339
- AtEndOfCommand 447
- automatic WAVE references 85
 - GetOperationDestWave 255
 - GetOperationWaveRef 252
 - SetOperationWaveRef 251

B

- background processing
 - SpinProcess 421

- backslashes
 - EscapeSpecialCharacters 429
 - UnEscapeSpecialCharacters 430
- balloon help 220
 - Igor Tips 220
- base SDK
 - in Xcode 37
- BCInt data type 191
- big-endian
 - MDChangeWave2 271
- bugs 225–235
- build location
 - Xcode 34
- bundles 4
 - Xcode 33
- byte order
 - MDChangeWave2 271
- byte reordering
 - settings 62
- C**
- C strings 108, 135
- C++
 - called from C code 218
 - calling C code 217
 - catch 218
 - converting from C 454
 - exceptions 218
 - linkage 454
 - templates 124
 - try 218
 - writing XOPs in 217
- CalcWaveRange 83, 266
- callbacks 9, 12
 - IGOR.lib 9
 - not allowed from private threads 172
 - thread-safe 171
 - threads 170–171
- CallFunction 214, 402
 - from an Igor thread 214
 - from an independent module 214
- Cancel button 208
- cancelling
 - CheckAbort 420
- Can't load the executable error 39
- Can't run Intel XOP error 39
- Capitalize 445
- Carbon
 - balloon help 220
 - history 3
 - paths 202
 - windows 153
 - XOP Toolkit 6 4
- catch
 - in C++ 218
- categories
 - for functions 100
 - of operations 71
- CatPossiblyQuotedName 388
- CFBundleExecutable 238
- CFM 3
 - resources 52
- changed XOPSupport routines
 - XOPGetFilePosition2 447
 - XOPNumberOfBytesInFile2 447
 - XOPSetFilePosition2 447
- ChangeWave 258
- chart displays
 - XOPSupport routines 378–379
- CheckAbort 216, 420
- CheckFunctionForm 214, 401
- CheckItem 486
- CheckName 386
- CheckRunningInMainThread 173, 406
- CheckTerm 447
- child data folder 303–304
- CHIO routines removed 447
- CHUNKS 125, 257, 280
- CLEANUP message 55
 - data folder reference counting 141
 - wave reference counting 132
- CleanupName 385
- CLEAR message 160
- CLEAR_MODIFIED message 56, 62
- CLICK message 158
 - not sent on Windows 155
- CLOSE message 157
- close type code for windows 158
- CMD message 70
 - error codes 57
- CmpStr 428
- CNTL resources 207
- colons 202
- color tables
 - GetIgorColorTableInfo 391
 - GetIgorColorTableValues 391
 - GetIndexedIgorColorTableName 390
 - GetNamedIgorColorTableHandle 390
 - XOPSupport routines 390–392
- column-major order 125
- COLUMNS 125, 257, 280
- command help
 - 64-bit XOPs 195
- command parsing routines
 - removed in XOP Toolkit 6 447
- command template 73
- command templates 74
 - length limit 93
- command window
 - HistoryDelete 393
 - HistoryDisplaySelection 393

- HistoryFetchParagraphText 394
- HistoryFetchText 395
- HistoryGetSelLocs 394
- HistoryInsert 393
- HistoryLines 393
- HistorySetSelLocs 394
- XOPSupport routines 393–396
- commands
 - adding 69–96
 - PauseUpdate 414
 - PutCmdLine 422
 - ResumeUpdate 414
 - XOPCommand 412
 - XOPCommand2 412
 - XOPCommand3 413
 - XOPSilentCommand 413
- commas
 - in operations 69
- communicating with Igor
 - XOPSupport routines 243–245
- compatibility 66–67
 - external operations 95
 - Igor 1.2 67
 - Igor Pro 67
 - of XOPs with Igor versions 4
 - operations 70
 - XOPI resource 53
- compilableOp operation category 71
- compiler version
 - LLVM 4
 - Xcode 37
- compiling
 - help files 219
- complex conjugate
 - example external function 97
- complex conjugate example 106
- complex numbers
 - external function example 97
 - parameters in external functions 106
 - results in external functions 106
 - storage in waves 125
- compound names 82
- computation threads 168
- ConcatenatePaths 340
- configurations
 - Visual C++ 44
- content region 163
- ConvertData 380
- ConvertData2 381
- COPY message 160
- CountInt data type 191
- CountMItems 486
- CR_STR
 - change required by Visual C++ 2015 Visual C++ 2015
 - CR_STR change required 487

- Create New XOP Project 21
- CreatePopupMenu 207, 333
- CreateValidDataObjectName 386
 - change in XOP Toolkit 6 450
- CreateWindowEx 154
- CreateXOPWindow 155
- CreateXOPWindowClass 155
- creating a project 21
- cross-platform development 201
 - dialogs 204
 - files 201
 - Macintosh emulation 435
- CtoToolStr
 - removed from XOP Toolkit 6 450
- current data folder 301
- cursors
 - ArrowCursor 362
 - HandCursor 362
 - IBeamCursor 362
 - setting 158
 - setting in NULLEVENT 158
 - setting on Macintosh 154
 - SpinCursor 362
 - WatchCursor 362
- curve fitting
 - function parameters 103
 - SimpleFit sample XOP 10, 18
 - SimpleGaussFit sample XOP 18
 - WAVE_TYPE 103
- custom errors 60
- CUT message 160

D

- dangling pointers 199, 232–233
- data acquisition
 - DoXOPIdle operation 177
 - from threads 174–179
 - IDLE message 177
 - mutexes 177
 - WaveHandleModified 178
- data folder reference counting
 - CLEANUP message 141
 - NEW message 141
 - structure parameters 141
- data folder reference parameters
 - external operations 84
- data folder reference waves 132
- data folder references
 - external function example 105
 - in structure parameters 210
 - parameters in external functions 104
 - results in external functions 104
- data folders 138–141
 - accessing
 - XOPSupport routines 298–313

- child 303–304
- commonly used XOPSupport routines 139
- conventions 140
- current 301
- DataFolderAndName example 87
- DuplicateDataFolder 305
- DuplicateDataFolderObject 311
- FetchWaveFromDataFolder 259
- free 303
- GetCurrentDataFolder 301
- GetDataFolderByIDNumber 302
- GetDataFolderIDNumber 298
- GetDataFolderListing 300
- GetDataFolderNameOrPath 298
- GetDataFolderObject 308
- GetDataFolderProperties 299
- GetIndexedChildDataFolder 304
- GetIndexedDataFolderObject 307
- GetNamedDataFolder 301
- GetNumChildDataFolders 303
- GetNumDataFoldersObjects 306
- GetParentDataFolder 303
- GetRootDataFolder 301
- GetWavesDataFolder 304
- HoldDataFolder 313
- ID numbers 298, 302
- KillDataFolder 305
- KillDataFolderObject 310
- listing 300
- MoveDataFolder 306
- MoveDataFolderObject 312
- NewDataFolder 304
- Packages data folder 140
- parent 303
- reference counting 140
- ReleaseDataFolder 313
- RenameDataFolder 306
- RenameDataFolderObject 312
- root 301
- SetCurrentDataFolder 301
- SetDataFolderObject 309
- SetDataFolderProperties 299
- ThreadGroupGetDF 407
- ThreadGroupPutDF 407
- threads 169
- variables 289
- waves 304
- data full-scale 262–263
 - MDGetWaveScaling 271
 - MDSetWaveScaling 272
- data sharing 65
- data types
 - 64-bit XOPs 190
 - BCInt 191
 - CountInt 191
 - IndexInt 191
 - PSInt 191
 - SInt16 190
 - SInt32 190
 - SInt64 190
 - SInt8 190
 - TickCountInt 190
 - UInt16 190
 - UInt32 190
 - UInt64 190
 - UInt8 190
 - XOPIORecParam 194
 - XOPIORecResult 193
- DATAFOLDER_OBJECT
 - CheckName 386
 - CreateValidDataObjectName 386
- DATAFOLDER_TYPE 122
 - data folder reference parameters 104
 - in XOPF resource 99
 - result in external functions 104
- dataFolderAndName keyword
 - Operation Handler 75
- DataFolderAndName parameters
 - example 85, 87
 - external operations 84
 - WAVE references 85
- DataFolderHandles 138
- dataFolderRef keyword
 - Operation Handler 75
- DataObjectValue structure 450
- DataObjectValuePtr
 - GetDataFolderObject 308
 - SetDataFolderObject 309
- dates 431–432
- DateToIgorDateInSeconds 431
- debug configuration
 - Visual C++ 44
- debugging 236–237
 - in Visual C++ 46
 - symbolic debugging 236
 - tricks 236
 - Xcode 40
 - XOPNotice 236
- default help file name 220
- DeleteMenuItem 486
- DeletePopMenuItems 335
- deleting waves
 - versus killing 129
- deployment target
 - in Xcode 37
- dereferencing a handle 233
- dereferencing a wave handle 234
- dereferencing handles 198
- destination waves 84–85
 - GetOperationDestWave 253
 - SetOperationWaveRef 251
- DestroyXOPWindow 155

- DEV_SYS_CODE 53
- development system 53
 - supported systems 31
- development systems
 - supported systems 4, 441
- DFREFs 84
 - data folder reference counting 141
 - in structure parameters 210
- dialogs
 - adding 203–209
 - AddPopMenuItems 334
 - Cancel button 208
 - CreatePopMenu 333
 - DeletePopMenuItems 335
 - DialogStorage structure 205
 - DisableDControl 329
 - DisplayDialogCmd 208, 332
 - DisposeDialogStorage 205
 - DisposeXOPDialog 326
 - dlgx resources 204
 - DLOG and DITL resources 204
 - Do It button 208
 - DoXOPDialog 325
 - EnableDControl 328
 - FillPathPopMenu 336
 - FillPopMenu 335
 - FillWavePopMenu 336
 - FillWindowPopMenu 337
 - FinishDialogCmd 208, 332
 - GetCheckBox 328
 - GetDBox 327
 - GetDDouble 330
 - GetDInt 329
 - GetDInt64 330
 - GetDText 329
 - GetPopMenu 335
 - GetPopMenuHandle 333
 - GetRadBut 327
 - GetXOPDialog 325
 - HandleItemHit 205
 - help 223
 - Help button 208–209
 - HiliteDControl 328
 - Igor style 208
 - IgorError 421
 - InitDialogSettings 205
 - InitDialogStorage 205
 - InitPopMenus 332
 - ItemIsPopMenu 333
 - kDialogFlagsUseControlHierarchy 204
 - KillPopMenus 337
 - not supported in Xcode6 487
 - on Macintosh 204
 - on Windows 204
 - popup menus 207, 209
 - resources 204
 - SelEditItem 331
 - SelMacEditItem 331
 - SetCheckBox 328
 - SetDDouble 331
 - SetDialogBalloonHelpID 326
 - SetDialogPort 325
 - SetDInt 330
 - SetDInt64 330
 - SetDText 329
 - SetPopItem 334
 - SetPopMatch 334
 - SetRadBut 327
 - ShowDialogWindow 325
 - ShutdownDialogSettings 205
 - To Clip button 208
 - To Cmd button 208
 - ToggleCheckBox 327
 - XOP_WINDOW_REF 204
 - XOPDialog 326
 - XOPDisplayHelpTopic 423
 - XOPOpenFileDialog 349
 - XOPSaveFileDialog 351
 - XOPSupport routines 325–337
- DialogStorage structure 205
- dimension labels
 - GetWaveDimensionLabels 275
 - MDGetDimensionLabel 274
 - MDSetDimensionLabel 274
 - SetWaveDimensionLabels 276
- dimension scaling
 - for waves 127
 - MDGetWaveScaling 271
 - MDSetWaveScaling 272
- dimensions
 - MDChangeWave 269
 - MDChangeWave2 271
 - MDGetWaveDimensions 269
 - MDMakeWave 267
 - ROWS, COLUMNS, LAYERS, CHUNKS 125, 257
- direct access method 124
- direct functions 114
 - resources 114, 116
- direct method for external functions 114
- direct method for external operations 70
- direct operations
 - resources 95
- DisableDControl 329
- DisableItem 486
- disabling menu items 150–152
- DisplayDialogCmd 208, 332
- DISPLAYSELECTION message 161
- DisposeDialogStorage 205
- DisposeHandle 438
- DisposePtr 436
- DisposeWindow 154

- DisposeXOPDialog 326
- DITL resources 203–204
- dlgx resources 204
- DLOG resources 203–204
- Do It button 208
- DoCHIO 445
- DoCHIO removed 447
- DoFunction routine 114
- DOITID 332
- DOUBLE
 - removed from XOP Toolkit 6 450
- double name parameters
 - external operations 82
 - independent modules 82
- DOUBLE_NAN 485
- doubleName keyword
 - Operation Handler 75
- DoUpdate 414
 - recursion 64
- DoWindowRecreationDialog 399
- DoXOPDialog 325
- DoXOPIdle operation 64, 177
 - data acquisition 177
- DUPLICATE message 161
- DuplicateDataFolder 305
- DuplicateDataFolderObject 311

E

- email 16
- EnableDControl 328
- EnableItem 486
- enabling menu items 150–152
- endian
 - MDChangeWave2 271
- error codes 57
 - from operations 70
- errors 57–61
 - custom error alert 60
 - custom errors 60
 - custom XOP error codes 58
 - error codes 57
 - external functions 102
 - FIRST_XOP_ERR 60
 - GetIgorErrorMessage 57, 60, 421
 - Igor error codes 58
 - IgorError 57, 60, 421
 - in external functions 116
 - Mac OS error codes 58
 - STR# 1100 resource 60
 - Windows OS error codes 58
 - WindowsErrorToIgorError 409–410
 - XOPOKAlert 60
 - XOPOKCancelAlert 60
 - XOPYesNoAlert 60
 - XOPYesNoCancelAlert 60
- escape sequences
 - EscapeSpecialCharacters 429
 - UnEscapeSpecialCharacters 430
- EscapeSpecialCharacters 429
- exceptions
 - in C++ 218
- ExecuteOperation 72
 - extern "C" declaration 92
 - NULL handles 92
- EXP_xxx 56–57
- experiment settings 56–57
- experiment type argument 56–57
- experiments 61–62
 - CLEAR_MODIFIED message 56
 - EXP_xxx 56–57
 - LOAD message 56
 - LOAD_TYPE_xxx 57
 - loading settings 62
 - LOADSETTINGS message 57
 - MODIFIED message 56
 - NEW message 56
 - SAVE message 56
 - SAVE_TYPE_xxx 56
 - SAVESETTINGS message 56
 - saving settings 61
- EXPORT_GRAPHICS message 161
- extended structure parameters 90
 - example in external operation 91
- extensions
 - help files 219
 - help files for 64-bit XOPs 195
- extern "C" declaration
 - calling C from C++ 217
 - ExecuteOperation 92
 - external functions 101
 - in C++ called from C 218
 - in C++ XOPs 454
 - with routines called by Igor 51
- external functions 97–117
 - adding 98
 - calling from an XOP 213
 - complex parameters 106
 - complex results 106
 - data folder reference parameters 104
 - data folder reference results 104
 - direct functions 114
 - direct method 114
 - DoFunction routine 114
 - error codes 102, 116
 - examples 97
 - extern "C" declaration 101
 - FUNCADDRS message 55, 114
 - function categories 100
 - FUNCTION message 114
 - FV_REF_TYPE 112
 - guided tour 18

- invoking from Igor 100
- keep RESIDENT 113
- logfit example 116
- message method 114
- names of 98, 100
- optional parameters 97
- parameter types 98, 102
- parameters 97, 101
- pass-by-reference 112
- pass-by-value 112
- resources 114, 116
- result types 98, 102
- results 102
- returning results 101
- SimpleFit sample XOP 10
- string parameters 106–108, 113
- string results 106–107
- structure alignment 101
- structure parameter example 109
- structure parameters 109
- thread-safe 114
- tp pointer 115
- versus user functions 97
- wave assignment statements 101
- wave parameters 103, 117
- wave results 103
- WaveAccess sample XOP 11
- XFUNC1 sample XOP 9
- XFUNC2 sample XOP 10
- XFUNC3 sample XOP 10
- external operations 69–96
 - as keyword 74
 - command template 73
 - command templates 74
 - compatibility 95
 - compatibility with Igor Pro 4 70
 - compilableOp 71
 - compilableOp operation category 71
 - data folder reference parameters 84
 - DataFolderAndName parameters 84
 - double name parameters 82
 - extended structure parameter example 91
 - extended structure parameters 90
 - flags 69, 73
 - GBLoadWave sample XOP 10
 - GetOperationDestWave 253
 - GetOperationWaveRef 252
 - keywords 73
 - kOperationIsThreadSafe 94
 - MenuXOP1 sample XOP 10
 - mnemonic names 74
 - name parameters 81
 - names of 72
 - NIGPIB2 sample XOP 11
 - Operation Handler 72–96
 - Operation Handler XOPSupport routines 246

- optional parameters 75, 79
- output variables 92
- parameters 69, 73
- RegisterOperation 246
- resources 95
- runtime parameter structure 77
- SimpleLoadWave sample XOP 10
- starter code 72
- string parameters 81
- structure parameter example 88
- structure parameters 87
- template 73
- templates 74
- thread-safe 93
- tp pointer 94
- TUDemo sample XOP 11
- variables 246–250, 355
- VarName parameters 83
- VDT2 sample XOP 11
- vs keyword 74
- wave parameters 82
- wave references 251–252
 - GetOperationDestWave 253
- WaveRange parameters 82
- WindowXOP1 sample XOP 10
- XOP1 sample XOP 9
- XOPop operation category 71

F

- F_EXTERNAL function category 100
- F_THREADSafe 115
- FetchNumericDataUsingVarName 84, 248
- FetchNumericValue 282
- FetchNumVar 290
 - example 137
- FetchStrHandle 292
- FetchStringDataUsingVarName 84, 250
- FetchStrVar 291
 - example 137
- FetchWave 259
- FetchWaveFromDataFolder 259
- fields
 - of structure parameters 210
- FIFOFileHeader
 - corrected spelling 450
- FIFOs
 - GetNamedFIFO 378
 - MarkFIFOUpdated 378
 - NamedFIFO.h 378
 - SoundInput XOP 378
 - XOPSupport routines 378–379
- FIFOSplitFileHeader
 - corrected spelling 450
- file loaders
 - FileLoaderMakeWave 354

- GBLoadWave sample XOP 10
- GetFullPathFromSymbolicPathAndFilePath 353
- Load Waves submenu 10
- output variables 92
- SanitizeWaveName 385
- SetFileLoaderOperationOutputVariables 92, 355
- SetFileLoaderOutputVariables 92, 354
- SimpleLoadWave sample XOP 10
- XOPOpenFileDialog 349
- XOPSaveFileDialog 351
- XOPSupport routines 353–356
- FILE_LOADER flags 354
- FileLoaderGetOperationFlags2 447
- FileLoaderMakeWave 354
 - change in XOP Toolkit 6 450
- filenames
 - illegal characters 338–339
- files
 - accessing
 - XOPSupport routines 338–352
 - ConcatenatePaths 340
 - cross-platform routines 201
 - FullPathPointsToFile 342
 - FullPathPointsToFolder 342
 - GetDirectoryAndFileNameFromFullPath 341
 - GetLeafName 342
 - GetNativePath 340
 - HFSToPosixPath 339
 - long names 448
 - MacToWinPath 338
 - MAX_DIRNAME_LEN 202
 - MAX_FILENAME_LEN 202
 - MAX_PATH_LEN 202
 - MAX_VOLUMENAME_LEN 202
 - ParseFilePath 343
 - path conversions 202
 - path separator characters 202
 - platform independence 10
 - SpecialDirPath 343
 - TUSFInsertFile 370
 - TUSFWriteFile 370
 - WinToMacPath 339
 - XOPAtEndOfFile 348
 - XOPCloseFile 345
 - XOPCreateFile 344
 - XOPDeleteFile 344
 - XOPGetFilePosition 347
 - XOPGetFilePosition2 348
 - XOPNumberOfBytesInFile 349
 - XOPNumberOfBytesInFile2 349
 - XOPOpenFile 344
 - XOPReadFile 345
 - XOPReadFile2 345
 - XOPReadFile64 346
 - XOPReadLine 346
 - XOPSetFilePosition 348
 - XOPSetFilePosition2 348
 - XOPWriteFile 347
 - XOPWriteFile64 347
- FillMenu 486
- FillMenuNoMeta 486
- FillPathMenu 486
- FillPathPopupMenu 336
- FillPopupMenu 335
- FillWaveMenu 486
- FillWavePopupMenu 336
- FillWindowPopupMenu 337
- FillWinMenu 486
- find
 - menu item 151, 316
 - TUFind 368
- FIND message 151, 160
- FinishDialogCmd 208, 332
- FIRST_XOP_ERR 58, 60
- FIRSTCMD 422
- FIRSTCMDCRHIT 422
- FixByteOrder 383
- flags 69, 73
 - optional parameters 75
 - prefix characters 74
- forward slashes 202
- frameworks
 - Xcode 33
- free data folders
 - GetParentDataFolder 303
- free waves 132
 - GetOperationDestWave 254
 - HoldWave 132
 - KillWave 133
 - MDMakeWave 268
 - reference counting 132
 - reference counting examples 133
 - ReleaseWave 132
- full size position message 159
- FullPathPointsToFile 342
- FullPathPointsToFolder 342
- FUNCADDRS message 55, 114
- FUNCREFs
 - CallFunction from an independent module 214
 - in structure parameters 210
 - using from an XOP 213
- function categories 100
 - F_EXTERNAL 100
- function index 114
- FUNCTION message 114
 - error codes 57
- functions
 - adding 97–117
 - CallFunction 214
 - calling from an XOP 213

CheckFunctionForm 214, 401
GetFunctionInfo 214, 399, 402
GetFunctionInfoFromFuncRef 214, 400
GetIgorCallerInfo 404
GetIgorRTStackInfo 404
FV_REF_TYPE 112
 in XOPF resource 99
FV_STRUCT_TYPE 109
 in XOPF resource 99

G

GBLoadWave
 porting to 64-bits 182
GBLoadWave sample XOP 10
GBLoadWave XOP
 Igor-style dialog 208
GET_TARGET_WINDOW_NAME message 162
GET_TARGET_WINDOW_REF message 162
GetActiveWindowRef 357
GetAString 447
GetAStringInHandle 447
GetCheckBox 328
GetCStringFromHandle 126, 135, 427
GetCurrentDataFolder 301
GetDataFolder 447
GetDataFolderAndName 447
GetDataFolderByIDNumber 302
GetDataFolderIDNumber 298
GetDataFolderListing 300
GetDataFolderNameOrPath 298
GetDataFolderObject 308
GetDataFolderProperties 299
GetDBox 327
GetDDouble 330
GetDInt 329
GetDInt64 330, 453
GetDirectoryAndFileNameFromFullPath 341
GetDLong 445, 453
GetDText 329
GetFlag 447
GetFlagNum 447
GetFormat 447
GetFullPathFromSymbolicPathAndFilePath 353
GetFunctionInfo 214, 399
GetFunctionInfoFromFuncRef 214, 400
GetHandleSize 108, 135, 437
 don't use on wave handles 265
 use with strings 237
GetIgorCallerInfo 404
GetIgorColorTableInfo 391
GetIgorColorTableValues 391
GetIgorErrorMessage 57, 60, 421
GetIgorProcedure 397
GetIgorProcedureList 397
GetIgorRTStackInfo 404
GetIndexedChildDataFolder 304
GetIndexedDataFolderObject 307
GetIndexedIgorColorTableName 390
GetKeyword 447
GetLeafName 342
GetLong 447
GetMenuHandle 149, 486
GetMenuID 486
getmenuitemtext 486
GetName 447
GetNamedDataFolder 301
 example 140
GetNamedFIFO 378
GetNamedIgorColorTableHandle 390
GetNativePath 203, 340
GetNewWindow 153
GetNum 447
GetNum2 447
GetNumChildDataFolders 303
GetNumDataFoldersObjects 306
GetNumVarName 447
GetNVAR 211, 295
 example 212
GetOperationDestWave
 example 85
GetParentDataFolder 303
GetPathInfo 445
GetPathInfo2 419
GetPopupMenu 207, 335
GetPopupMenuHandle 333
GetPrefsState 61, 426
GetPtrSize 435
GetRadBut 327
GetRootDataFolder 301
 example 140
GetStrVarName 447
GetSVAR 211, 296
 example 212
GetSymb 447
GetTextWaveData 126, 285
GetTrueOrFalseFlag 447
GetWave 447
GetWaveDimensionLabels 275
GetWaveList 447
GetWaveName 447
GetWaveRange 447
GetWavesDataFolder 304
GetWavesInfo 445
GetXOPDialog 325
GetXOPIndString 376
GetXOPItem 51, 54, 245
 items field in IORecHandle 63
 pitfalls 237
 replace long data type 194
GetXOPMessage 51, 54, 244
 message field in IORecHandle 63

- pitfalls 237
- replace long data type 194
- GetXOPNamedResource 376
- GetXOPPrefsHandle 61, 425
- GetXOPRefCon 445
- GetXOPResource 376
- GetXOPResult 244
- GetXOPStatus 245
- GetXOPWindow 153, 357
- GetXOPWindowIgorPositionAndState 361
- GetXOPWindowPositionAndState 359
- global variables 225
 - threads 168
- GPIB communications
 - NIGPIB2 sample XOP 11
- GRAF_MASK 424
- GROW message 157
 - not sent on Windows 155

H

- HandAndHand 437
- HandCursor 362
- HandleItemHit 205
- Handles
 - disposing 81
 - GetCStringFromHandle 126, 135
 - PutCStringInHandle 126, 135
 - WaveMemorySize 265
- handles 66, 197–201
 - containing strings 237
 - dangling pointers 233
 - dereferencing 198, 233
 - locking 199
 - master pointers 198
 - recommended practices 235
 - remove Handle locking code 452
 - resizing 199
 - treating as C strings 108, 135
 - usage 199
- HandToHand 437
- hdlg resources 220, 451
- header files 7
 - changes in XOP Toolkit 6 443
 - IgorErrors.h 443
- heap 198
 - fragmentation 199
 - overwriting 226
- help 219–223
 - balloon help 220
 - command help 27
 - for 64-bit XOPs 195
 - for dialogs and windows 223
 - for operations and functions 220
 - function categories 100
 - help file name 220
 - Igor Help Browser 220
 - Igor tips for menu items 221
 - operation categories 71
 - searching for help files 423
 - status line help 222
 - STR# 1101 resource 220
 - technical support 15
 - tips 424
 - topics and subtopics 423
 - XOPDisplayHelpTopic 223, 423
 - XOPSetContextualHelpMessage 424
- Help Browser 27
- Help button 208
 - in dialogs 209
- help files
 - compiling 219
 - extensions 219
 - for 64-bit XOPs 195
 - location of 219
- HFS
 - paths 202, 338
- HFS paths 448
- HFSToPosixPath 203, 339
- HGetState 199, 445
 - obsolete 200, 452
- HideAndDeactivateXOPWindow 358
- HideXOPWindow 358
- hiding
 - windows 164
- HiliteDControl 328
- history
 - cistory carbon copy 416
 - XOPNotice 414
 - XOPNotice2 415
 - XOPNotice3 416
 - XOPResNotice 417
- history area
 - HistoryDelete 393
 - HistoryDisplaySelection 393
 - HistoryFetchParagraphText 394
 - HistoryFetchText 395
 - HistoryGetSelLocs 394
 - HistoryInsert 393
 - HistoryLines 393
 - HistorySetSelLocs 394
- HistoryDelete 393
- HistoryDisplaySelection 393
- HistoryFetchParagraphText 394
- HistoryFetchText 395
- HistoryGetSelLocs 394
- HistoryInsert 393
- HistoryLines 393
- HistorySetSelLocs 394
- HLock 199, 445
 - obsolete 200, 452

HMNU resource
 status line help 222
 HMNU resources
 status line help 222
 hmnu resources 220, 451
 HMODULE 409
 HoldDataFolder 140, 313
 HoldWave 129, 287
 free waves 132
 wave reference waves 132
 HOST_IMPORT
 in C++ 456
 HSetState 199, 445
 obsolete 200, 452
 HSTRING_TYPE 107, 113
 in XOPF resource 99
 HUnlock 199, 445
 obsolete 200, 452
 HWND 157
 IgorClientHWND 409
 XOP_WINDOW_REFS 153

I

IBeamCursor 362
 ID numbers for data folders 298, 302
 IDLE message 54
 data acquisition 177
 recursion 64–65
 IDLE messages
 DoXOPIdle operation 64, 177
 IDLES bit 54, 63
 SetXOPType 243
 idling 63
 IEEE_FLOAT 380
 Igor 1.2
 compatibility 67
 Igor commands 412–413, 422
 Igor Extensions folder 5, 11, 70, 98
 64-bit XOPs 181
 Igor Help Browser 220
 Igor mailing list 16
 Igor number type codes 380, 382
 Igor Pro
 compatibility 67, 215
 first Windows version 3
 Igor Extensions folder 5
 Igor Pro 3.1 3
 Igor Pro 4 3
 Igor Pro 5 Windows compatibility 3
 Igor Pro 6 3
 Igor Pro 6.1 3
 Igor Pro 6.2 3
 Igor Pro 6.20 215
 Igor Pro 6.3 4
 Igor User Files folder 5
 Igor Pro 4
 external operations 70
 Igor Pro User Files
 SpecialDirPath 343
 Igor Pro User Files folder
 64-bit XOPs 181
 Igor Tips 220
 XOPSetContextualHelpMessage 424
 Igor tips
 for menu items 221
 STR# resources 221
 WindowXOP1 XOP example 221
 Igor User Files folder 5
 Igor version 16
 Igor window coordinates 162
 Igor-style dialogs 208
 Igor-thread-safe 170
 IGR_OBSOLETE 66, 452
 IGR.lib
 callback routines 9
 link with 237
 memory management routines 197, 435
 XOPSupport 7
 IGR32 181
 IGR32 #define 443
 IGR64 181
 IGR64 #define 443
 IGR64.lib 188
 IgorClientHWND 409
 IgorColorSpec 391
 IgorDateInSecondsToDate 432
 IgorError 57, 60, 421
 IgorErrors.h 443
 IgorExchange 16
 IgorModule 409
 IgorVersion 423
 igorVersion 66
 igorVersion global 66
 IgorXOP.h 7
 IgorXOPs 6 folder
 copy existing XOPs to 442
 IgorXOPs6 folder
 contents of 6
 creating 5
 IGR0 XOP file creator 11, 49
 Xcode 39
 illegal characters 338–339
 INDENTLEFT message 161
 INDENTRIGHT message 161
 independent modules
 CallFunction 214
 double name parameters 82
 IndexInt data type 191
 Info.plist
 Xcode 33
 Info.plist file 39, 238

InfoPlist.strings file 39
INFs 430
INIT message 12, 54
 error codes 57
InitDialogSettings 205
InitDialogStorage 205
initialization 11–12, 54, 243
InitPopMenus 207, 332
INSERTCMD 422
INSERTFILE message 161
insertmenuitem 486
installing XOP Toolkit 5
Intel 3
 compiling in Xcode 23
 initial support of 3
Interface Builder 52, 204
internal threads 168
 callbacks not allowed 172
 data acquisition 177–178
IORecHandle 12, 49, 51, 62–64
 passed to main 15
 recursion 65
IsINF32 430
IsINF64 430
IsMacOSX 445
IsNaN32 431
IsNaN64 431
IsStringExpression 447
IsXOPWindowActive 357
ITEM_REQUIRES bits 150
itemFlags field 150
ItemIsPopMenu 333
IXOP XOP file type 11, 49
 Xcode 39

K

kDialogFlagsUseControlHierarchy 204
KEY message 158
 not sent on Windows 155
Keyword 447
keywords 73
 in operations 69
 optional parameters 75
 prefix characters 74
KillDataFolder 305
KillDataFolderObject 310
killing waves 127
 preventing killing 128
 versus deleting 129
KillPopMenus 207, 337
KillWave 259
 free waves 133
kMDWaveAccessMode0 124, 276
kOpDestWaveChangeExistingWave 254
kOpDestWaveMakeFreeWave 254

kOpDestWaveMustAlreadyExist 254
kOpDestWaveOverwriteExistingWave 254
kOpDestWaveOverwriteOK 254
kOperationIsThreadSafe 94

L

LAYERS 125, 257, 280
leaks 231
Legendre polynomials 10, 97
liberal names 384
 CatPossiblyQuoteName 388
 CleanupName 385
 CreateValidDataObjectName 386
 PossiblyQuoteName 388
libraries
 link errors 237
 XOPSupport 7
libXOPSupport.a 33
linkage
 in C++ XOPs 454
linking
 errors 237
little-endian
 MDChangeWave2 271
LLVM compiler 4
 main function issue 215
LNK2112 error 187
LNK4099 warnings 46
LOAD message 56
Load Waves submenu 10
LOAD_TYPE_xxx 57
loading settings 62
LOADSETTINGS message 57, 61–62
lock state
 waves 262
locking handles 199
logfit
 example external function 10, 97, 116
long data type
 64-bit XOPs 190, 192
 replacing 193–194
long file names 448

M

Mac OS X
 first Igor version 3
Mach-O 3
 creating projects in Xcode 32
 resources 52
MACIGOR #define 443
Macintosh
 paths 202
 supported OS versions 4
 XOP Toolkit 6 requirements 4
Macintosh Emulation 435

- Macintosh emulation 4
 - DisposeHandle 438
 - DisposePtr 436
 - GetHandleSize 437
 - GetPtrSize 435
 - HandAndHand 437
 - HandToHand 437
 - MemError 439
 - memory management 66, 197
 - memory management routines 435–439
 - menus
 - XOPSupport routines 439
 - NewHandle 436
 - NewPtr 435
 - PtrAndHand 439
 - PtrToHand 439
 - routines 435–440
 - SetHandleSize 438
 - SetPtrSize 436
 - TickCount 440
- Macintosh menu manager routines 143
- MacRectToWinRect 434
- MacToWinPath 203, 338
- mailing list 16
- main 12
 - called by Igor 49
 - LLVM compatibility 215
 - receives INIT message 54
 - receives IORECHandle 15
 - version checking 66
 - XOPMain 215
- main function 12
 - changed in XOP Toolkit 6 451
- main thread 168
- MAIN_NAME_SPACE 384
- MakeWave 257
 - change in XOP Toolkit 6 450
- MallocDebug
 - leaks 231
- MarkFIFOUpdated 378
- master pointers 198
- MAX_DIM_LABEL_CHARS 274–276
- MAX_DIRNAME_LEN 202, 448
- MAX_FILENAME_LEN 202, 448
- MAX_PATH_LEN 202
- MAX_UNIT_CHARS 263–264, 273
- MAX_VOLUMENAME_LEN 202
- MAXWAVES
 - removed from XOP Toolkit 6 450
- MDAccessNumericWaveData 124, 276
- MDChangeWave 269
- MDChangeWave2 271
- MDGetDimensionLabel 274
- MDGetDPDataFromNumericWave 123, 280
- MDGetNumericWavePointValue 123, 278
- MDGetTextWavePointValue 283
 - example 126
- MDGetWaveDimensions 125, 269
- MDGetWaveScaling 271
- MDGetWaveUnits 273
- MDI
 - child windows 154–155
 - destroying a window 155
 - getting Igor client HWND 409
 - getting window position 359
 - menu bar 156
 - MOVE_TO_FULL_POSITION message 159
 - RETRIEVE message 160
 - setting window position 163, 360
 - window coordinates 360
- MDMakeWave 267
 - example 121
- MDSetDimensionLabel 274
- MDSetNumericWavePointValue 123, 278
- MDSetTextWavePointValue 283
 - example 126
- MDSetWaveScaling 127, 272
 - example 121
- MDSetWaveUnits 127, 273
- MDStoreDPDataInNumericWave 123, 280
- MemClear 427
- MemError 439
- memory
 - 64-bit XOPs 181
 - checking allocations 231
 - dangling pointers 232–233
 - data sharing 65
 - DisposeHandle 438
 - DisposePtr 436
 - disposing 230–231
 - GetHandleSize 437
 - GetPtrSize 435
 - HandAndHand 437
 - handles 197–201
 - HandToHand 437
 - heap 198
 - leaks 230–231
 - Macintosh emulation 66, 197
 - Macintosh memory management 197–201
 - MemError 439
 - NewHandle 436
 - NewPtr 435
 - pointers 198
 - PtrAndHand 439
 - PtrToHand 439
 - relocatable block 198–199
 - SetHandleSize 438
 - SetPtrSize 436
 - using unallocated blocks 229
- memory management routines 435–439
- IGOR.lib 435

- menu handles 149
- menu IDs 148–149
 - conflicts 152
 - ranges 152
- menu management routines 439
- menu manager routines 143
- MENU resources
 - for popup menus 207
- MENUENABLE message 54, 150, 154
- MENUHELP resources
 - status line help 222
- MENUITEM message 54, 148
 - error codes 57
 - on Windows 156
 - recursion 64
- menus
 - ActualToResourceItem 150, 315
 - ActualToResourceMenuID 149, 314
 - adding 143–152
 - adding a main menu 146
 - adding a main menu item 146–148
 - dialog popup menus 207
 - disabling 150–152
 - enabling 150–152
 - enabling for XOP window 154
 - enabling Igor items 151
 - Find item 151, 316
 - getting menu handles 149
 - Igor tips 221
 - limitations 145
 - menu ID conflicts 152
 - menu ID ranges 152
 - menu IDs 148–149
 - menu manager routines 143
 - MENUENABLE message 54
 - MENUITEM message 54, 148
 - MenuXOP1 sample XOP 10, 143, 145
 - popup menus 209
 - resource IDs 148
 - resources 145–148
 - ResourceToActualItem 149, 315
 - ResourceToActualMenuID 149, 314
 - responding to 148–150
 - SetIgorMenuItem 315
 - SHOW_MENU_AT_LAUNCH 146
 - SHOW_MENU_WHEN_ACTIVE 146
 - status line help 222
 - TUFixEditMenu 369
 - TUFixFileMenu 369
 - Undo item 316
 - WM_COMMAND message 148
 - WM_INITMENU message 150
 - XMI1 1100 resource 145–146
 - XMN1 1100 resource 145–146
 - XOPActualMenuIDToMenuRef 316
 - XOPAppendMenuItem 320
 - XOPCheckMenuItem 322
 - XOPCountMenuItems 317
 - XOPDeleteMenuItem 320
 - XOPDeleteMenuItemRange 321
 - XOPDisableMenuItem 321
 - XOPEnableMenuItem 321
 - XOPFillMenu 315
 - XOPFillMenuNoMeta 323
 - XOPFillPathMenu 323
 - XOPFillWaveMenu 323
 - XOPFillWinMenu 324
 - XOPGetMenuInfo 317
 - XOPGetMenuItemInfo 318
 - XOPGetMenuItemText 319
 - XOPHideMainMenu 318
 - XOPInsertMenuItem 320
 - XOPMenuRef 145
 - XOPResourceMenuIDToMenuRef 316
 - XOPSetMenuItemText 319
 - XOPShowMainMenu 318
 - XOPSupport routines 314–324
 - XSM1 1100 resource 145, 147
- MenuXOP1 sample XOP 10
- MenuXOP1 XOP 143, 145
- message method for external functions 114
- messages
 - ACTIVATE message 157
 - basic 54–55
 - between Igor and XOP 63
 - CLEANUP message 55
 - CLEAR message 160
 - CLEAR_MODIFIED message 56
 - CLICK message 158
 - CLOSE message 157
 - COPY message 160
 - CUT message 160
 - DISPLAYSELECTION message 161
 - DUPLICATE message 161
 - EXPORT_GRAPHICS message 161
 - FIND message 151, 160
 - for XOPs with windows 156–162
 - from Igor to XOP 12
 - FUNCADDRS message 55, 114
 - GET_TARGET_WINDOW_NAME message 162
 - GET_TARGET_WINDOW_REF message 162
 - GROW message 157
 - IDLE message 54
 - INDENTLEFT message 161
 - INDENTRIGHT message 161
 - INIT message 12, 54
 - INSERTFILE message 161
 - KEY message 158
 - LOAD message 56
 - LOADSETTINGS message 57, 61–62
 - MENUENABLE message 54, 154

MENUITEM message 54, 148
 MODIFIED message 56
 MOVE_TO_FULL_POSITION message 159
 MOVE_TO_PREFERRED_POSITION message 159
 NEW message 56, 61
 NULLEVENT message 154, 158
 OBJINUSE 128
 OBJINUSE message 55
 PAGESETUP message 161
 PASTE message 160
 PRINT message 161
 recursion 65
 REPLACE message 161
 RETRIEVE message 160
 REVERT_WINDOW message 162
 SAVE message 56
 SAVE_WINDOW message 162
 SAVE_WINDOW_AS message 162
 SAVE_WINDOW_COPY message 162
 SAVE_WINDOW_MACRO message 162
 SAVEFILE message 161
 SAVESETTINGS message 56, 61
 saving and loading settings 56–57
 SELECT_ALL message 161
 SET_TARGET_WINDOW_NAME message 162
 SET_TARGET_WINDOW_TITLE message 162
 SETGROW message 157
 threads 168
 UNDO message 160
 UPDATE message 154, 157
 WINDOW_MOVED message 158
 XOP_HIDE_WINDOW message 158
 XOP_SHOW_WINDOW message 159
 XOPEntry routine 12
 meta-characters 324
 XOPFillMenu 315
 XOPFillMenuNoMeta 323
 Microsoft 3
 miscellaneous XOPSupport routines 412–426
 mlong-branch warning 455
 mnemonic names 74
 modification count
 waves 261
 modification date
 waves 261
 modification state
 waves 261
 MODIFIED message 56
 MOVE_TO_FULL_POSITION message 159
 MOVE_TO_PREFERRED_POSITION message 159
 MoveDataFolder 306
 MoveDataFolderObject 312
 MoveHHi
 obsolete 200, 452

MoveLockHandle 199, 445
 obsolete 201, 452
 Multitasking
 see threads
 Multithreading 167
 see also threads
 mutexes 169
 data acquisition 177
 MW_MASK 424
 My Mac 32-bit 35
 My Mac 64-bit 35

N

name keyword
 Operation Handler 75
 name parameters
 external operations 81
 NamedFIFO.h 378
 names
 CatPossiblyQuotedName 388
 CheckName 386
 CleanupName 385
 CreateValidDataObjectName 386
 liberal 385–386, 388
 of external functions 100
 of external operations 72
 PossiblyQuoteName 388
 SanitizeWaveName 385
 standard 388
 UniqueName 384
 UniqueName2 384
 WaveName 265
 NaNs 431
 National Instruments
 NIGPIB2 sample XOP 11
 native paths 202
 NEW message 56, 61
 data folder reference counting 141
 wave reference counting 132
 new XOPSupport routines
 CheckRunningInMainThread 449
 FetchNumericDataUsingVarName 449
 FetchStringDataUsingVarName 449
 GetDInt64 449
 GetOperationDestWave 449
 GetOperationWaveRef 449
 HoldDataFolder 449
 HoldWave 449
 ParseFilePath 448
 ReleaseDataFolder 449
 ReleaseWave 449
 RunningInMainThread 449
 SetDInt64 449
 SetOperationStrVar2 449
 SpecialDirPath 448

- TUFetchText2 449
- XOPGetFilePosition2 448
- XOPNumberOfBytesInFile 448
- XOPReadFile64 448
- XOPSetFilePosition2 448
- XOPWriteFile64 448
- NewDataFolder 304
 - example 140
- NewHandle 198, 436
 - example 199
- NewPtr 198, 435
 - example 199
- NextSymb 447
- nibs 52, 204
- NIGPIB2 sample XOP 11
- no scheme problem
 - Xcode 36
- NT_CMLPX 106
 - in XOPF resource 99
 - number type for waves 122
- NT_FP32 257, 380
 - number type for waves 122
- NT_FP64 257, 380
 - in XOPF resource 99
 - number type for waves 122
- NT_I16 257, 380
 - number type for waves 122
- NT_I32 257, 380
 - number type for waves 122
- NT_I8 257, 380
 - number type for waves 122
- NT_UNSIGNED 257, 380
 - number type for waves 122
- NULLEVENT message 154, 158
 - not sent on Windows 155
- number keyword
 - Operation Handler 75
- number type codes 380, 382
- number types
 - for waves 122
 - GBLoadWave example 10
- NumBytesAndFormatToNumType 382
- numeric conversion
 - ConvertData 380
 - ConvertData2 381
 - FixByteOrder 383
 - NumBytesAndFormatToNumType 382
 - NumTypeToNumBytesAndFormat 382
 - ScaleClipAndRoundData 383
 - ScaleData 382
- numeric conversion routines 380–383
 - GBLoadWave example 10
- numeric expressions
 - in operations 69
- numeric precision
 - in external functions 102

- NumTypeToNumBytesAndFormat 382
- NVAR 83
- NVARs
 - GetNVAR 295
 - in structure parameters 210, 212
 - SetNVAR 296

O

- object names
 - XOPSupport routines 384–389
- object types
 - WAVE_OBJECT 55
- objects
 - counting in data folders 306
 - duplicating in data folders 311
 - getting in data folders 307–308
 - killing in data folders 310, 312
 - renaming in data folders 312
 - setting in data folders 309
- OBJINUSE message 55, 128
- obsolete XOPSupport routines
 - HGetState 452
 - HLock 452
 - HSetState 452
 - HUnlock 452
 - MoveHHi 452
 - MoveLockHandle 452
- OLD_IGOR 452
- operation categories 71
 - compilableOp 71
 - XOPOp 71
- Operation Handler 72–96
 - introduced in Igor Pro 5 3
 - RegisterOperation 246
 - XOPSupport routines 246
- operations
 - adding 69–96
- optional parameters 75, 79
 - as keyword 76
 - in external functions 97
 - vs keyword 76
- output variables 92

P

- packages 4
 - .rsrc files 52
 - Xcode 32
- Packages data folder 140
- PAGESETUP message 161
- PANEL_MASK 424
- parameter types
 - in external functions 102
 - numeric precision 102
- parameters 73
 - as keyword 74

- checking string parameters in external functions 108
- commas between parameters 69
- complex in external functions 106
- data folder reference in external functions 104
- data folder reference parameters 84
- DataFolderAndName parameters 84
- disposing string parameters in external functions 108
- double name parameters 82
- external functions 101
- FV_REF_TYPE 112
- in external functions 97–98
- in operations 69
- name parameters 81
- optional 75
- optional parameters 79
- pass-by-reference 112
- pass-by-value 112
- runtime parameter structure 77
- string in external functions 106–107, 113
- string parameters 81
- structures 210
- structures in external functions 109
- structures in external operations 87
- threads 169
- type checking in external functions 99
- VarName parameters 83
- vs keyword 74
- wave in external functions 103, 117
- wave parameters 82
- WaveRange parameters 82
- parent data folder 303
- ParseFilePath 343
- ParseOperationTemplate 72
- parsing commands
 - GetFullPathFromSymbolicPathAndFilePath 353
- pass-by-reference 112
 - FV_REF_TYPE 112
- pass-by-value 112
- PASTE message 160
- path separator characters 202
- PathList 418
- paths
 - Carbon 202
 - ConcatenatePaths 340
 - escape characters 202
 - FullPathPointsToFile 342
 - FullPathPointsToFolder 342
 - GetDirectoryAndFileNameFromFullPath 341
 - GetFullPathFromSymbolicPathAndFilePath 353
 - GetLeafName 342
 - GetNativePath 203, 340
 - HFS 202, 338
 - HFSToPosixPath 339
 - illegal characters 338–339
 - Mac/Win conversion 202
 - Macintosh 202
 - MacToWinPath 203, 338
 - native paths 202
 - ParseFilePath 343
 - POSIX 202, 338
 - Posix 203
 - POSIX paths 202
 - separator 338
 - SpecialDirPath 343
 - WinToMacPath 203, 339
- PauseUpdate 414
- pdb files 46
- phone number for technical support 16
- pitfalls 237–239
- PL_MASK 424
- platform-independence 4
- platforms
 - x64 platform 182
- plgndr
 - example external function 10, 97
- point access method 123
- pointers 198
 - dangling pointers 232–233
 - misconceptions 228
- popup menus 209
 - AddPopMenuItems 334
 - CreatePopMenu 207, 333
 - DeletePopMenuItems 335
 - enabling and disabling 208
 - FillPathPopMenu 336
 - FillPopMenu 335
 - FillWavePopMenu 336
 - FillWindowPopMenu 337
 - GetPopMenu 207, 335
 - GetPopMenuHandle 333
 - in dialogs 207
 - InitPopMenus 207, 332
 - ItemIsPopMenu 333
 - KillPopMenus 207, 337
 - Macintosh 207
 - SetPopItem 334
 - SetPopMatch 334
- POSIX 202
 - paths 202, 338
- Posix 203
- PossiblyQuoteName 388
- predefined macros
 - _MACINTOSH_ 442
 - _WINDOWS_ 442
 - XOP Toolkit 6 443
- preferences 61
 - GetPrefsState 426
 - GetXOPPrefsHandle 425

- SaveXOPPrefsHandle 425
- SpecialDirPath 343
- structures 61
- prefix characters 74
- prefix files
 - Xcode 33, 38
- PRINT message 161
- printf
 - 64-bit XOPs 195
- private threads 168
 - callbacks not allowed 172
 - data acquisition 177–178
- problems 15
- procedures
 - CallFunction 214
 - calling from an XOP 213
 - CheckFunctionForm 214, 401
 - DoWindowRecreationDialog 399
 - GetFunctionInfo 214, 399, 402
 - GetFunctionInfoFromFuncRef 214, 400
 - GetIgorCallerInfo 404
 - GetIgorProcedure 397
 - GetIgorProcedureList 397
 - GetIgorRTStackInfo 404
 - SetIgorProcedure 398
 - XOPSupport routines 397–405
- programming utility routines 427–434
- project database files 46
- project files 8
 - Visual C++ 44
- project settings
 - Visual C++ 44
 - Xcode 37
- Project Settings dialog
 - Xcode 34
- projects
 - Create New XOP Project 21
 - creating in guided tour 21
 - creating in Visual C++ 43
 - creating in Xcode 32
 - debug and release configurations 44
- Property List Editor 238
- PSInt data type 191
- PtrAndHand 439
- PtrToHand 439
- PutCmdLine 422
- PutCStringInHandle 126, 135, 427

Q

- qualified keywords 80
- queues
 - thread output queue 175, 177
- quotes
 - CatPossiblyQuoteName 388
 - PossiblyQuoteName 388

R

- recent windows 164
- rectangles 434
- recursion 64–65, 162
 - DoUpdate side-effect 414
 - IDLE messages 64–65
 - XOPCommand side-effect 412
 - XOPCommand2 side-effect 412
 - XOPCommand3 side-effect 413
 - XOPSilentCommand side-effect 413
- reference counting
 - data folder reference waves 132
 - data folders 140
 - free wave examples 133
 - free waves 132
 - wave reference waves 132
 - waves 128
- RegisterClass 154
- RegisterFunction
 - replace long data type 193
- RegisterOperation 15, 70, 72, 246
 - example 49
- release configuration
 - Visual C++ 44
- ReleaseDataFolder 140, 313
 - data folder reference waves 132
- ReleaseWave 129, 288
 - free waves 132
 - wave reference waves 132
- relocatable block of memory 198–199
- RenameDataFolder 306
- RenameDataFolderObject 312
- REPLACE message 161
- REPLACEALLCMDs 422
- REPLACEALLCMDSCRHIT 422
- REPLACEFIRSTCMD 422
- ResEdit 52, 203
- RESIDENT bit 63
 - SetXOPType 243
- resident XOPs
 - definition of 12
 - external functions 113
 - XOPType field in IORec 63
- resource files 8
 - Visual C++ 44
- resource.h file
 - creation of 52
 - dialog resource IDs 204
 - menu resource IDs 146
- ResourceMenuIDToMenuHandle 486
- resources 51–53
 - CNTL 207
 - creating 52
 - creating on Windows 52
 - DITL 203

- DITL resources 204
- dlgx 204
- DLOG 203
- DLOG resources 204
- for dialogs 204
- for menus 145–148
- GetXOPIndString 376
- GetXOPNamedResource 376
- GetXOPResource 376
- hdlg resources 220
- HMNU resources 222
- hmnu resources 220
- in CFM XOPs 52
- in direct external functions 114, 116
- in direct external operations 95
- in Mach-O XOPs 52
- initialization 11
- MENU 207
- MENUHELP resources 222
- popup menus 207
- resource.h file 52, 146, 204
- status line help 221–222
- STR# 1100 resource 51, 58, 60
- STR# 1101 resource 51, 220
- STR# 1160 resource 51
- STR# for Igor tips 221
- version resources 209
- XMI1 1100 145–146
- XMI1 1100 resource 51, 221
- XMN1 1100 145–146
- XMN1 1100 resource 51
- XOP-specific 51
- XOPC 1100 resource 51, 70
- XOPF 1100 resource 51, 98
- XOPI 1100 resource 51–52
- XOPRefNum 376
- XOPSupport routines 376–377
- XOPTypes.r 52
- XPRF resources 425
- XSM1 1100 145, 147
- XSM1 1100 resource 51
- ResourceToActualItem 149, 315
- ResourceToActualMenuID 149, 314
- responding to menu selections 148–150
- result codes 57
- result types
 - in external functions 102
- results
 - external functions 101–102
 - string in external functions 106–107
- ResumeUpdate 414
- RETRIEVE message 160
- REVERT_WINDOW message 162
- Rez
 - compiling .r files 52
- RGBColor 391

- root data folder 301
 - threads 169
- routine descriptors
 - XOPDialog 326
- ROWS 125, 257, 280
- RS232
 - VDT2 sample XOP 11
- RunningInMainThread 173, 406
- runtime library
 - Visual C++ 45
- runtime parameter structure 77

S

- S_ variables
 - in external operations 92
- S_fileName 354–355
 - SetFileLoaderOperationOutputVariables 355
- S_path 354–355
 - SetFileLoaderOperationOutputVariables 355
- S_waveNames 354–355
 - SetFileLoaderOperationOutputVariables 355
- sample XOPs 8–11
- SanitizeWaveName 385
- SAVE message 56
- SAVE_TYPE_xxx 56
- SAVE_WINDOW message 162
- SAVE_WINDOW_AS message 162
- SAVE_WINDOW_COPY message 162
- SAVE_WINDOW_MACRO message 162
- SAVEFILE message 161
- SAVESETTINGS message 56, 61
- SaveXOPPrefsHandle 61, 425
- saving settings 61
- ScaleClipAndRoundData 383
- ScaleData 382
- scaling
 - waves 126
- schemes
 - Xcode 35
- SDK
 - in Xcode 37
 - Xcode 40
- SELECT_ALL message 161
- SelEditItem 331
- SelMacEditItem 331
- SendWinMessageToIgor 155, 410
- serial port
 - VDT2 sample XOP 11
- SET_TARGET_WINDOW_NAME message 162
- SET_TARGET_WINDOW_TITLE message 162
- SetCheckBox 328
- SetCurrentDataFolder 301
- SetDataFolderObject 309
- SetDataFolderProperties 299
- SetDDouble 331

- SetDialogBalloonHelpID 326
- SetDialogPort 325
- SetDInt 330
- SetDInt64 330, 453
- SetDLong 445, 453
- SetDText 329
- SetFileLoaderOperationOutputVariables 92, 355
- SetFileLoaderOutputVariables 92, 354
- SETGROW message 157
- SetHandleSize 199, 438
 - dangling pointers 234
 - use with strings 237
- SetIgorComplexVar 294
- SetIgorFloatingVar 293
 - example 137
- SetIgorIntVar 293
- SetIgorMenuItem 151, 315
- SetIgorProcedure 398
- SetIgorStringVar 294
 - example 137
- SetMenuID 486
- setmenuitemtext 486
- SetNaN32 431
- SetNaN64 431
- SetNVAR 211, 296
 - example 212
- SetOperationNumVar 246
- SetOperationStrVar 247
- SetOperationStrVar2 247
- SetOperationWaveRef 85, 251
- SetPopItem 334
- SetPopMatch 334
- SetPtrSize 436
- SetRadBut 327
- SetSVAR 211, 297
 - example 212
- SetTextWaveData 126, 287
- settings
 - byte reordering 62
 - GetPrefsState 426
 - GetXOPPrefsHandle 425
 - loading 62
 - messages for saving and loading settings 56–57
 - SaveXOPPrefsHandle 425
 - saving 61
 - saving and loading 61
 - structure alignment 62
 - structures 62
- SetWaveDimensionLabels 276
- SetWaveLock 262
- SetWaveNote 264
- SetWaveScaling 263
- SetWavesStates 445
 - obsolete 201
- SetWaveUnits 264
- SetXOPEntry 15, 243
 - example 49
 - XOPEntry field in IORecHandle 63
- SetXOPMessage 244
- SetXOPRefCon 445
- SetXOPResult 57, 244
 - example 49
 - replace long data type 193
 - result field in IORecHandle 64
 - returning custom errors 60
 - SAVESETTINGS message 61
- SetXOPType 243
 - IDLE message 54
 - in external functions 113
 - XOPType field in IORec 63
- SetXOPWindowIgorPositionAndState 361
- SetXOPWindowPositionAndState 360
- SetXOPWindowTitle 359
- shortcuts 11
- SHOW_MENU_AT_LAUNCH 146
- SHOW_MENU_WHEN_ACTIVE 146
- ShowAndActivateXOPWindow 358
- ShowDialogWindow 325
- showing
 - windows 164
- ShowXOPWindow 357
- ShutdownDialogSettings 205
- SIGNED_INT 380
- SimpleFit sample XOP 10, 18
- SimpleGaussFit
 - compiling 26
 - testing 27
- SimpleGaussFit sample XOP 18
- SimpleLoadWave sample XOP 10
- SINGLE_NAN 485
- SInt16 data type 190
- SInt32 data type 190
- SInt64 data type 190
- SInt8 data type 190
- Slow Data Acq example 176
- solution files 44
- SoundInput XOP 378
- SpecialDirPath 343
- SpinCursor 362
- SpinProcess 216, 421
 - recursion 64
- sprintf
 - 64-bit XOPs 195
- SS_MASK 424
- stack
 - overwriting 226
- standard names 388
- starter code 72
 - details 92
 - updating 93

- status line
 - XOPSetContextualHelpMessage 424
- status line help
 - for menus 222
- StoreNumericDataUsingVarName 83, 249
- StoreNumericValue 282
- StoreNumVar 291
- StoreStringDataUsingVarName 83, 250
- StoreStrVar 292
- STR# 1100 resource 51, 58, 60
- STR# 1101 resource 51, 220
 - XOP help file name 220
- STR# 1160 resource 51
- STR# resource
 - status line help 221
- STR# resources for Igor tips 221
- STR_OBJECT
 - CheckName 386
 - CreateValidDataObjectName 386
- strchr2 428
- String
 - in structure parameters 210
- string expressions
 - in operations 69
- string keyword
 - Operation Handler 75
- string parameters
 - external operations 81
- string variables
 - accessing
 - XOPSupport routines 289–297
 - content stored in handles 135
 - FetchStrHandle 292
 - FetchStringDataUsingVarName 250
 - FetchStrVar 291
 - GetSVAR 296
 - SetFileLoaderOperationOutputVariables 355
 - SetIgorStringVar 294
 - SetOperationStrVar 247
 - SetOperationStrVar2 247
 - SetSVAR 297
 - StoreStringDataUsingVarName 250
 - StoreStrVar 292
 - StringList 420
 - Variable 289
 - VarNameToDataType 248
- StringList 420
- strings
 - disposing 81
 - GetCStringFromHandle 126, 135
 - GetHandleSize 237
 - in handles 237
 - in Igor 108, 135
 - in structure parameters 211
 - pitfalls 237
 - PutCStringInHandle 126, 135
 - SetHandleSize 237
- strchr2 428
- Struct Member Alignment
 - Visual C++ 444
- structure alignment 77, 209
 - changing for XOP Toolkit 6 443
 - in external functions 101
 - settings structures 62
 - Visual C++ 444
- structure keyword
 - Operation Handler 75
- structure parameters 210
 - CheckFunctionForm 401
 - data folder reference counting 141
 - example in external function 109
 - example in external operation 88
 - extended 90
 - extended example in external operation 91
 - fields 210
 - in external functions 109
 - in external operations 87
 - NVARs 212
 - strings 211
 - SVARs 212
 - wave reference counting 130
- structures
 - expandable 61–62
 - GetNVAR 295
 - GetSVAR 296
 - in XOPF resource 99
 - SetNVAR 296
 - SetSVAR 297
 - versioning 90, 111
 - wave reference counting 88, 112, 212
- subclassing 156
- submenus
 - XSM1 1100 resource 147
- SVAR 83
- SVARs
 - GetSVAR 296
 - in structure parameters 210, 212
 - SetSVAR 297
- symbolic debugging 236
- symbolic paths
 - FillPathPopupMenu 336
 - GetFullPathFromSymbolicPathAndFilePath 353
 - GetPathInfo2 419
 - PathList 418
 - XOPFillPathMenu 323

T

- target extension
 - Visual C++ 45

- target machine
 - Visual C++ 46
- target name
 - Visual C++ 45
- target windows
 - adding 164
- TARGET_RT_MAC_MACHO 443, 459
- Task Manager
 - leaks 231
- technical support 15
 - email 16
 - mailing list 16
- TellIgorWindowsStatus 164
- TellIgorWindowStatus 358
- template 73
- templates 74
 - access waves 124
 - for operations and functions 220
 - length limit 93
- temporary storage access method 123
- text utility routines 363–375
 - removed in XOP Toolkit 6 446
 - TUDemo sample XOP 11
- text waves
 - accessing text data 126
 - faster access 126
 - GetTextWaveData 285
 - MDGetTextWavePointValue 283
 - MDSetTextWavePointValue 283
 - SetTextWaveData 287
 - text wave type 122
- text windows
 - TUActivate 364
 - TUClear 367
 - TUClick 367
 - TUCopy 366
 - TUCut 366
 - TUDelete 372
 - TUDemo sample XOP 11
 - TUDisplaySelection 364
 - TUDispose 363
 - TUDrawWindow 370
 - TUFetchParagraphText 373
 - TUFetchSelectedText 373
 - TUFetchText2 374
 - TUFind 368
 - TUFixEditMenu 369
 - TUFixFileMenu 369
 - TUGetDocInfo 371
 - TUGetSelLocs 371
 - TUGrow 364
 - TUIIdle 365
 - TUIndentLeft 369
 - TUIndentRight 369
 - TUInsert 372
 - TUKey 368
 - TULines 371
 - TUMoveToFullSizePosition 365
 - TUMoveToPreferredPosition 365
 - TUNew2 363
 - TUNull 366
 - TUPageSetupDialog 367
 - TUPaste 366
 - TUPrint 367
 - TUReplace 368
 - TURetrieveWindow 365
 - TUSelectAll 368
 - TUSetSelLocs 372
 - TUSetStatusArea 375
 - TUSFInsertFile 370
 - TUSFWriteFile 370
 - TUUndo 367
 - TUUpdate 364
- TEXT_WAVE_TYPE 122, 257
 - type for waves 122
- thread groups 176, 178
- thread output queue 175, 177
- thread-safe
 - callbacks 171
 - external functions 114
 - external operations 93
 - Igor-thread-safe 170
 - programming 167–168
 - with Igor Pro 6.13 or later 172
 - with old versions of Igor 171
- ThreadGroupGetDF 177, 407
- ThreadGroupGetDFR operation 175
- ThreadGroupPutDF 177, 407
- ThreadGroupPutDF operation 175
- threading
 - Visual C++ 44
- threads 167
 - callbacks 170–171
 - not allowed from private threads 172
 - CallFunction 214
 - CheckRunningInMainThread 173, 406
 - computation threads 168
 - data acquisition 174–179
 - data folders 169
 - DoXOPIdle operation 177
 - external operations 93
 - global variables 168
 - IDLE message 177
 - Igor-thread-safe 170
 - internal 168
 - callbacks not allowed 172
 - internal threads 177–178
 - main 168
 - messages from Igor 168
 - mutexes 169
 - data acquisition 177
 - parameters 169

- private 168
 - callbacks not allowed 172
 - private threads 177–178
 - root data folder 169
 - RunningInMainThread 173, 406
 - Slow Data Acq example 176
 - thread groups 176, 178
 - thread output queue 175, 177
 - ThreadGroupGetDF 407
 - ThreadGroupGetDF callback 177
 - ThreadGroupGetDFR operation 175
 - ThreadGroupPutDF 407
 - ThreadGroupPutDF callback 177
 - ThreadGroupPutDF operation 175
 - ThreadStart operation 169
 - tp pointer 94, 115
 - types of 168
 - user-defined function threads 168–169
 - UserFunctionThreadInfoPtr 94, 115
 - WaveHandleModified 178
 - waves 169
 - with Igor Pro 6.13 or later 172
 - with old versions of Igor 171
 - XOPNotice 236
 - XOPSupport routines 406–408
 - threadSafeOp 94
 - ThreadStart operation 169
 - TickCount 440
 - TickCountInt data type 190
 - To Clip button 208
 - To Cmd button 208
 - TOCLIPID 332
 - TOCMDID 332
 - ToggleCheckBox 327
 - TooltoCStr
 - removed from XOP Toolkit 6 450
 - tp pointer 94, 115
 - TransformWindowCoordinates 360
 - TRANSIENT bit 63
 - SetXOPType 243
 - transient XOPs
 - definition of 12
 - with windows 155
 - XOPType field in IORec 63
 - troubleshooting 15
 - try
 - in C++ 218
 - TU windows 156
 - subclassing 156
 - TUDispose 156
 - TUNew2 156
 - window procedure 156
 - TUActivate 364
 - TUClear 367
 - TUClick 367
 - TUCopy 366
 - TUCut 366
 - TUDelete 372
 - TUDemo sample XOP 11
 - TUDisplaySelection 364
 - TUDispose 156, 363
 - TUDrawWindow 370
 - TUFetchParagraphText 373
 - TUFetchSelectedText 373
 - TUFetchText 446
 - TUFetchText2 374
 - TUFind 368
 - TUFixEditMenu 369
 - TUFixFileMenu 369
 - TUGetDocInfo 371
 - TUGetSelLocs 371
 - TUGetText 446
 - TUGrow 364
 - TUIIdle 365
 - TUIndentLeft 369
 - TUIndentRight 369
 - TUInsert 372
 - TUInsertFile 446
 - TUKey 368
 - TULength 446
 - TULines 371
 - TULoc structure 372
 - TUMoveToFullSizePosition 365
 - TUMoveToPreferredPosition 365
 - TUNew 446
 - TUNew2 156, 363
 - TUNull 366
 - TUPageSetupDialog 367
 - TUPaste 366
 - TUPrint 367
 - TUReplace 368
 - TURetrieveWindow 365
 - TUSelectAll 368
 - TUSelectionLength 446
 - TUSelEnd 446
 - TUSelStart 446
 - TUSetSelect 446
 - TUSetSelLocs 372
 - TUSetStatusArea 375
 - TUSFInsertFile 370
 - TUSFWriteFile 370
 - TUUndo 367
 - TUWriteFile 446
 - type cast conversion error
 - Visual C++ 2015 487
 - types of XOPs 12
- ## U
- UInt16 data type 190
 - UInt32 data type 190
 - UInt64 data type 190

- UInt8 data type 190
- UNC
 - volume names 338–339
- undo
 - menu item 316
 - TUUndo 367
- UNDO message 160
- UnEscapeSpecialCharacters 430
- uninitialized variables 226
- UniqueName 384
- UniqueName2 384
- units 263–264
 - for waves 127
 - MDGetWaveUnits 273
 - MDSetWaveUnits 273
 - waves 126
- UNSIGNED_INT 380
- UPDATE
 - recursion 65
- update
 - DoUpdate 414
 - XOPCommand side-effect 412
 - XOPCommand2 side-effect 412
 - XOPCommand3 side-effect 413
 - XOPSilentCommand side-effect 413
- UPDATE message 154, 157
 - not sent on Windows 155
 - recursion 64
- updates for XOP Toolkit 31
- user functions
 - versus external functions 97
- user-defined function threads 168–169
- user-defined functions
 - calling from an XOP 213
- UserFunctionThreadInfoPtr 94, 115
- utility routines 9, 427
 - CheckRunningInMainThread 406
 - CmpStr 428
 - DateToIgorDateInSeconds 431
 - EscapeSpecialCharacters 429
 - GetCStringFromHandle 427
 - IgorDateInSecondsToDate 432
 - IsINF32 430
 - IsINF64 430
 - IsNaN32 431
 - IsNaN64 431
 - MacRectToWinRect 434
 - MemClear 427
 - PutCStringInHandle 427
 - RunningInMainThread 406
 - SetNaN32 431
 - SetNaN64 431
 - strchr2 428
 - strrchr2 428
 - UnEscapeSpecialCharacters 430
 - WinInfo 424

- WinRectToMacRect 434
- XOPOKAlert 433
- XOPOKCancelAlert 433
- XOPYesNoAlert 433
- XOPYesNoCancelAlert 433

V

- V_ variables
 - in external operations 92
- V_Flag
 - SetFileLoaderOperationOutputVariables 355
- V_flag 354–355
- VAR_GLOBAL 289
- VAR_OBJECT
 - CheckName 386
 - CreateValidDataObjectName 386
- Variable 289
 - in structure parameters 210
- VariableList 419
- variables
 - accessing
 - XOPSupport routines 289–297
 - accessing data in 135–137
 - commonly used XOPSupport routines 136
 - data folders 289
 - example of making 137
 - FetchNumericDataUsingVarName 248
 - FetchNumVar 290
 - FetchStrHandle 292
 - FetchStringDataUsingVarName 250
 - FetchStrVar 291
 - GetNVAR 295
 - GetSVAR 296
 - getting in external operations 248, 250
 - local versus global 289
 - making global 137
 - precision of 289
 - SetFileLoaderOperationOutputVariables 355
 - SetIgorComplexVar 294
 - SetIgorFloatingVar 293
 - SetIgorIntVar 293
 - SetIgorStringVar 294
 - SetNVAR 296
 - SetOperationNumVar 246
 - SetOperationStrVar 247
 - SetOperationStrVar2 247
 - SetSVAR 297
 - setting in external operations 246–250, 355
 - StoreNumericDataUsingVarName 249
 - StoreNumVar 291
 - StoreStringDataUsingVarName 250
 - StoreStrVar 292
 - StringList 420
 - uninitialized 226
 - VAR_GLOBAL 289

- Variable 289
- VariableList 419
- VarNameToDataType 248
- varName keyword
 - Operation Handler 75
- VarName parameters
 - external operations 83
- VarNameToDataType 83, 248
- vc90.pdb 46
- VDT2 sample XOP 11
- version
 - of Igor 66
- version checking 66
- version resources 209
- versions 66–67
 - checking Igor version 452
 - compatibility 4
 - of Igor 4, 16, 70
 - of structures 90, 111
 - XOP Toolkit 6 Upgrade 441
- virtual memory
 - 64-bit XOPs 181
- Visual C++ 43–47
 - .pdb files 46
 - .rc files 44
 - .sln files 44
 - .vcxproj files 44
 - 64-bit XOPs 182
 - changing projects for XOP Toolkit 6 460
 - configurations 44
 - converting project to VC2015 47
 - converting to Visual C++ 2015 460
 - creating a new project 43
 - debug configuration 44
 - debugging 46
 - first XOP support 3
 - LNK2112 error 187
 - LNK4099 46
 - pdb files 46
 - porting to 64-bits 182
 - project database files 46
 - project files 44
 - project settings 44
 - release configuration 44
 - resource files 44
 - resources 52
 - runtime library 45
 - solution files 44
 - structure alignment 210, 444
 - supported versions 4, 31
 - target extension 45
 - target machine 46
 - target name 45
 - testing installation 6
 - threading 44
 - vc90.pdb 46

- warnings 46
- worksheet files 8
- x64 platform 182
- XOPSupport project files 7
- Visual C++ 10
 - support for 441
- Visual C++ 2008
 - support for 441
- Visual C++ 2010
 - support for 441
- Visual C++ 2012
 - building SimpleFit 20
- Visual C++ 2015
 - projects added in XOP Toolkit 6.40 487
 - type cast conversion error 487
- Visual C++ 9
 - building SimpleFit 20
 - support for 441
 - testing installation 6
- volume names
 - UNC 338–339
- vs keyword 74
 - optional parameters 76

W

- warnings
 - Wno-parentheses 38
 - Visual C++ 46
- WatchCursor 362
- wave assignment statements 101
- wave handles
 - dereferencing 234
- wave keyword
 - Operation Handler 75
- wave parameters
 - external operations 82
- wave reference counting 128
 - CLEANUP message 132
 - NEW message 132
 - structure parameters 88, 112, 130, 212
- wave reference waves 132
- WAVE references
 - data folder reference counting 130
 - DataFolderAndName parameters 85
 - destination wave example 85
 - GetOperationDestWave 253
 - in external operations 251–252
 - in structure parameters 210
 - SetOperationWaveRef 85
- wave references
 - automatic 85
 - GetOperationDestWave 255
 - GetOperationWaveRef 252
 - SetOperationWaveRef 251
- wave scaling 126

- wave units 126
- WAVE_OBJECT
 - CheckName 386
 - CreateValidDataObjectName 386
- WAVE_OBJECT object type 55
- WAVE_TYPE 103, 122
 - curve fitting 103
 - in XOPF resource 99
 - result in external functions 103
 - wave reference parameters 103
- WaveAccess sample XOP 11, 124
- WaveData 265
 - dangling pointers 234
 - example 121
- WaveHandleModified 267
 - threads 178
- WaveHandlesModified 445
- WaveList 417
- WaveLock 262
- WaveMemorySize 265
- WaveModCount 261
- WaveModDate 261
- WaveModified 267
- WaveModState 261
- WaveName 265
- WaveNote 264
- WavePoints 260
- waveRange keyword
 - Operation Handler 75
- WaveRange parameters
 - external operations 82
- waves
 - accessing
 - XOPSupport routines 257–288
 - accessing data in 119–134
 - accessing numeric data 122–125
 - accessing text data 126
 - accessing with C++ templates 124
 - CalcWaveRange 266
 - ChangeWave 258
 - commonly used XOPSupport routines 119
 - containing data folder references 132
 - containing wave references 132
 - data folders 304
 - data scaling 127
 - destination wave
 - SetOperationWaveRefs 251
 - destination waves 85
 - GetOperationDestWave 253
 - dimension scaling 127
 - direct access method 124
 - example making 121
 - FetchNumericValue 282
 - FetchWave 259
 - FetchWaveFromDataFolder 259
 - FileLoaderMakeWave 354
 - FillWavePopupMenu 336
 - free 132
 - GetOperationDestWave 254
 - MDMakeWave 268
 - GetOperationDestWave 253
 - GetOperationWaveRef 252
 - GetTextWaveData 285
 - GetWaveDimensionLabels 275
 - HoldWave 287
 - killing 127
 - killing versus deleting 129
 - KillWave 259
 - lock state 262
 - MakeWave 257
 - MDAccessNumericWaveData 276
 - MDChangeWave 269
 - MDChangeWave2 271
 - MDGetDimensionLabel 274
 - MDGetDPDataFromNumericWave 280
 - MDGetNumericWavePointValue 278
 - MDGetTextWavePointValue 283
 - MDGetWaveDimensions 269
 - MDGetWaveScaling 271
 - MDGetWaveUnits 273
 - MDMakeWave 267
 - MDSetDimensionLabel 274
 - MDSetNumericWavePointValue 278
 - MDSetTextWavePointValue 283
 - MDSetWaveScaling 272
 - MDSetWaveUnits 273
 - MDStoreDPDataInNumericWave 280
 - number types 122
 - OBJINUSE message 55
 - organization of numeric data 125
 - parameters in external functions 103
 - point access method 123
 - preventing killing 128
 - ranges of 82
 - reference counting 128
 - ReleaseWave 288
 - results in external functions 103
 - SanitizeWaveName 385
 - scaling 126
 - SetFileLoaderOutputVariables 354
 - SetOperationWaveRef 251
 - SetTextWaveData 287
 - SetWaveDimensionLabels 276
 - SetWaveLock 262
 - SetWaveNote 264
 - SetWaveScaling 263
 - SetWaveUnits 264
 - speed of accessing numeric data 124
 - StoreNumericValue 282
 - temporary storage access method 123
 - text wave type 122
 - threads 169

- units 126–127
- WaveAccess example 11
- WaveData 265
- WaveHandleModified 267
- WaveList 417
- WaveLock 262
- WaveMemorySize 265
- WaveModCount 261
- WaveModDate 261
- WaveModified 267
- WaveModState 261
- WaveName 265
- WaveNote 264
- WavePoints 260
- WaveScaling 262
- WaveType 260
- WaveUnits 263
- XOPFillWaveMenu 323
- XOPWaveAccess.c 7
- WaveScaling 262
- WaveType 260
- WaveUnits 263
- Win32 platform 182
- window procedures 155
 - for TU windows 156
- WINDOW_MOVED message 158
 - not sent on Windows 155
- WindowPtr 157
 - XOP_WINDOW_REFS 153
- windows
 - activating 157, 164
 - adding 153–165
 - adding on Macintosh 153
 - adding on Windows 154
 - ArrowCursor 362
 - captions 359
 - clearing 160
 - click events 158
 - close type code 158
 - closing 157
 - content region 163
 - coordinates 163
 - copying 160
 - CreateWindowEx 154
 - CreateXOPWindow 155
 - CreateXOPWindowClass 155
 - creating on Macintosh 153
 - cursors 158
 - cutting 160
 - destroying an MDI window 155
 - DestroyXOPWindow 155
 - disposing on Macintosh 154
 - DUPLICATE message 161
 - EXPORT_GRAPHICS message 161
 - FillWindowPopupMenu 337
 - finding 160
 - GET_TARGET_WINDOW_NAME message 162
 - GET_TARGET_WINDOW_REF message 162
 - GetActiveWindowRef 357
 - getting Igor client HWND 409
 - GetXOPWindow 357
 - GetXOPWindowIgorPositionAndState 361
 - GetXOPWindowPositionAndState 359
 - HandCursor 362
 - help 223
 - hide message 158
 - HideAndDeactivateXOPWindow 358
 - HideXOPWindow 358
 - hiding 164
 - IBeamCursor 362
 - Igor window coordinates 162
 - indenting 161
 - inserting files in 161
 - IsXOPWindowActive 357
 - key events 158
 - MDI child window menu bar 156
 - MDI child windows 154–155
 - MDI window coordinates 360
 - MDI window position 163, 359–360
 - messages for 156–162
 - move to full position message 159
 - move to preferred position message 159
 - moved message 158
 - not supported in Xcode6 487
 - null events 158
 - page setup 161
 - pasting 160
 - printing 161
 - recent 164
 - recursion problems 162
 - RegisterClass 154
 - replacing 161
 - resizing 157, 364
 - retrieve message 160
 - REVERT_WINDOW message 162
 - SAVE_WINDOW message 162
 - SAVE_WINDOW_MACRO message 162
 - saving files from 161
 - SELECT_ALL message 161
 - SendWinMessageToIgor 155, 410
 - SET_TARGET_WINDOW_NAME message 162
 - SET_TARGET_WINDOW_TITLE message 162
 - SetXOPWindowIgorPositionAndState 361
 - SetXOPWindowPositionAndState 360
 - SetXOPWindowTitle 359
 - show message 159
 - ShowAndActivateXOPWindow 358
 - showing 164
 - ShowXOPWindow 357
 - SpinCursor 362

- target windows 164
- TellIgorWindowsStatus 164
- TellIgorWindowStatus 358
- text
 - XOPSupport routines 363–375
- TransformWindowCoordinates 360
- transient XOPs 155
- TU windows 156
- TUNew2 156
- undoing 160
- updating 157
- WatchCursor 362
- window procedures 155
- windowKind field 153, 357
- WindowXOP1 sample XOP 10
- WinList 418
- XOP_WINDOW_REF 157
- XOPFillWinMenu 324
- XOPSetContextualHelpMessage 424
- XOPSupport routines 357–362
- XOPWindowProc 155
- zooming 157, 364

Windows OS

- IgorClientHWND 409
- IgorModule 409
- SendWinMessageToIgor 410
- WindowsErrorToIgorError 410
- WMGetLastError 409
- XOPModule 409

Windows OS error codes 58

Windows platform

- compatibility with Igor Pro 5 3
- first Igor release 3
- supported OS versions 4
- XOPSupport routines 409–411

WindowsErrorToIgorError 410

WindowXOP1 sample XOP 10

WindowXOP1 XOP

- Igor tips example 221

WINIGOR #define 443

WinInfo 424

WinList 418

WinRectToMacRect 434

winresrc.h 451

WinToMacPath 203, 339

WINVER 450

WM_ACTIVATE message 154

WM_CHAR message 154–155

- in TU windows 156

WM_COMMAND message 148, 154, 156

WM_DESTROY message 155

WM_INITMENU message 150, 154

WM_KEY message 154–155

WM_LBUTTONDOWN message 155

WM_MDIACTIVATE message 155

- in TU windows 156

WM_MDIDESTROY message 155

WM_MOVE message 155

WM_PAINT message 154–155

- in TU windows 156
- recursion 64

WM_RBUTTONDOWN message 155

WM_SIZE message 155

WMDeleteMenu 486

WMDeleteMenuItems 486

WMDrawMenuBar 486

WMGetLastError 409

WMGetMenuFromModule 486

WMInsertMenu 486

WMTypes.h 192

WMTypes.h file 190

WNAME_SIZE

- removed from XOP Toolkit 6 450

worksheet files 8

World-Wide Web 16

X

X scaling 262–263

- MDGetWaveScaling 271
- MDSetWaveScaling 272

x64 platform 182

x86 3

Xcode 31

- .exp files 32, 38
- base SDK 37
- build location 34
- bundles 33
- changing projects for XOP Toolkit 6 457
- command help 21
- compiler version 37
- compiling for Intel 23
- converting project to Xcode 41
- creating a new project 32
- creating project in guided tour 23
- debugging 40
- deployment target 37
- executable name 39
- frameworks 33
- IGR0 XOP file creator 39
- Info.plist 33
- Info.plist file 39
- InfoPlist.strings file 39
- IXOP XOP file type 39
- LLVM compiler 4
- Mach-O 32
- no scheme problem 36
- packages 32
- prefix files 33, 38
- project settings 34, 37
- Project Settings dialog 34
- schemes 35

- SDK 37, 40
 - support for 441
 - supported operating systems 4
 - supported versions 4, 31
 - testing installation 6
- XOPSupport project files 7
- Xcode 4
 - building SimpleFit 18
 - compiler versionr 4
 - LLVM 4
- Xcode 6
 - dialogs not supported 487
 - projects added in XOP Toolkit 6.40 487
 - windows not supported 487
- XFUNC1 sample XOP 9
- XFUNC1 XOP
 - description 97
 - direct functions 114
 - XOPF 1100 resource example 98
- XFUNC1Add
 - example external function 97
- XFUNC1ComplexConjugate
 - example external function 97, 106
- XFUNC1Div
 - example external function 97
 - invoking 100
 - parameter types 102
 - parameters and result 101
- XFUNC2 sample XOP 10
- XFUNC2 XOP
 - description 97
 - direct functions 114
 - XOPF 1100 resource 102
- XFUNC3 sample XOP 10
- XFUNC3 XOP
 - description 97
 - XOPF 1100 resource 106
- XMI1 1100 resource 51, 145–146
 - adding Igor tips 221
 - itemFlags field 150
 - status line help 221
- XMN1 1100 resource 51, 145–146
- XOP errors 58
- XOP files 11
- XOP protocol 11–12
 - compatibility 66
 - GetXOPItem 245
 - GetXOPMessage 244
 - GetXOPResult 244
 - GetXOPStatus 245
 - SetXOPEntry 243
 - SetXOPMessage 244
 - SetXOPResult 244
 - SetXOPTYPE 243
 - version 53
 - XOPInit 243
 - XOP protocol version 67
 - XOP resources 51–53
- XOP Toolkit
 - installing 5
 - overview 6–9
 - Release 6.00 483
 - Release 6.01 483
 - Release 6.40 485
 - sample XOPs 8
 - updates 31
 - version 53
 - version 6 4
 - version 6 upgrade 441
- XOP Toolkit 6 3
 - 64-bit XOPs 181–182, 453
 - changed XOPSupport routines 447
 - changes for 64-bit XOPs 453
 - changes to main function 451
 - header file changes 443
 - miscellaneous changes 450
 - new file routines 448
 - new XOPSupport routines 448
 - obsolete XOPSupport routines 445
 - porting existing XOPs to 442
 - programming for 64-bit XOPs 191
 - removed command parsing routines 447
 - removed text utility routines 446
 - removed XOPSupport routines 445
 - renamed XOPSupport routines 447
 - supported Igor versions 452
- XOP Toolkit 6.3 4
- XOP Toolkit 6.4 4
- XOP Toolkit number type codes 380, 382
- XOP_DIALOG_REF
 - dialogs 204
- XOP_HIDE_WINDOW message 158
- XOP_SHOW_WINDOW message 159
- XOP_TOOLKIT_VERSION 53
- XOP_VERSION 53, 67
- XOP_WINDOW_REF 157
- XOP_WINDOW_REFS 153
- XOP.h 7
- XOP1 sample XOP 9
- XOPActualMenuIDToMenuRef 316
- XOPAppendMenuItem 320
- XOPAtEndOfFile 348
- XOPC 1100 resource 49, 51, 70
- XOPCheckMenuItem 322
- XOPCloseFile 345
- XOPCommand 412
 - recursion 64–65
- XOPCommand2 412
- XOPCommand3 413
- XOPCountMenuItems 317
- XOPCreateFile 344
- XOPDeleteFile 344

- XOPDeleteMenuItem 320
- XOPDeleteMenuItemRange 321
- XOPDialog 326
- XOPDisableMenuItem 321
- XOPDisplayHelpTopic 223, 423
- XOPEnableMenuItem 321
- XOPEntry 12, 15, 51, 54, 63
 - extern "C" declaration 51
 - FUNCTION message 114
 - messages 12
 - replace long data type 193
- XOPF 1100 resource 51, 114
 - complex parameters 106
 - data folder reference parameters 104
 - defined 98
 - inspected by Igor Pro 98
 - string parameters 106
 - wave parameters 103
 - XFUNC2 XOP example 102
- XOPFillMenu 315
- XOPFillMenuNoMeta 323
- XOPFillPathMenu 323
- XOPFillWaveMenu 323
- XOPFillWinMenu 324
- XOPGetFilePosition 347
- XOPGetFilePosition2 348
 - changed in XOP Toolkit 6 447
- XOPGetMenuInfo 317
- XOPGetMenuItemInfo 318
- XOPGetMenuItemText 319
- XOPHideMainMenu 318
- XOPI 1100 resource 49, 51–52
 - XOP protocol version 67
- XOPInit 15, 243
 - example 49
 - XOPRecHandle global 62
- XOPInsertMenuItem 320
- XOPIORecParam data type 194
- XOPIORecResult data type 193
- XOPMain 215
 - receives IORecHandle 15
- XOPMain function 12
 - changed in XOP Toolkit 6 451
- XOPMenuRef 145
- XOPModule 409
- XOPNavOpenFileDialog 448
- XOPNavSaveFileDialog 448
- XOPNotice 414
 - for debugging 236
 - from a thread 236
 - pitfalls 237
- XOPNotice2 415
- XOPNotice3 416
- XOPNumberOfBytesInFile 349
- XOPNumberOfBytesInFile2 349
 - changed in XOP Toolkit 6 447
- XOPOKAlert 60, 433
- XOPOKCancelAlert 60, 433
- XOPOp operation category 71
- XOPOpenFile 344
- XOPOpenFileDialog 203, 349
 - HFS paths 448
- XOPReadFile 345
- XOPReadFile2 345
- XOPReadFile64 346
- XOPReadLine 346
- XOPRecHandle global 62
- XOPRefNum 376
- XOPResNotice 417
- XOPResourceMenuIDToMenuRef 316
- XOPs
 - activating 5
- XOPS 1100 resource 61
- XOPSaveFileDialog 203, 351
 - HFS paths 448
- XOPSetContextualHelpMessage 424
- XOPSetFilePosition 348
- XOPSetFilePosition2 348
 - changed in XOP Toolkit 6 447
- XOPSetMenuItemText 319
- XOPShowMainMenu 318
- XOPSilentCommand 413
 - recursion 64–65
- XOPStandardHeaders.h 7
 - changes in XOP Toolkit 6 443
- XOPStructureAlignmentTwoByte.h 77
- XOPStructureAlignmentTwoBytes.h
 - removed in XOP Toolkit 6 443
- XOPSupport
 - callbacks 9, 12
 - header file changes 443
 - LNK4099 warnings 46
 - miscellaneous changes 450
 - new file routines 448
 - new routines 448
 - removed routines 445
 - renamed routines 447
 - supported Igor versions 452
 - utility routines 9
 - XOPSupport folder 9
- XOPSupport 6
 - changed routines 447
- XOPSupport folder 7, 9
- XOPSupport project files 7
- XOPSupport routines 242–440
 - color table routines 390–392
 - command window 393–396
 - for accessing data folders 298–313
 - for accessing files 338–352
 - for accessing variables 289–297
 - for accessing waves 257–288
 - for communicating with Igor 243–245

- for dialogs 325–337
- for FIFOs 378–379
- for file loaders 353–356
- for menus 314–324
- for text windows 363–375
- for windows 357–362
- GetDInt64 453
- GetDLong 453
- miscellaneous 412–426
- multithreading support routines 406–408
- numeric conversion 380–383
- object names 384–389
- Operation Handler 246
- procedures 397–405
- programming utilities 427–434
- resources 376–377
- SetDInt64 453
- SetDLong 453
- TUFetchText 446
- TUGetText 446
- TUInsertFile 446
- TULength 446
- TUNew 446

- TUSelectionLength 446
- TUSelEnd 446
- TUSelStart 446
- TUSetSelect 446
- TUWriteFile 446
- Windows-specific routines 409–411
- XOPSupport.c 7
- XOPSupport.h 7
- XOPSupport64.lib 188
- XOPTypes.r 52
- XOPWaveAccess.c 7
- XOPWindowProc 155
- XOPWriteFile 347
- XOPWriteFile64 347
- XOPYesNoAlert 60, 433
- XOPYesNoCancelAlert 60, 433
- XPRF resources 425
- XSM1 1100 resource 51, 145, 147
- xstrcat
 - example external function 97, 107
 - XOPF 1100 resource 106
- xstrcat0 external function 10
- xstrcat1 external function 10

