

# IGOR XOP TOOLKIT

---

## REFERENCE MANUAL

Version 7.03

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: September 27, 2018 (7.03)

© Copyright 2018 WaveMetrics, Inc. All rights reserved.

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

Email: [support@wavemetrics.com](mailto:support@wavemetrics.com), [sales@wavemetrics.com](mailto: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 7.....	4
Choosing Which XOP Toolkit to Use .....	5
Macintosh and Windows XOPs .....	6
Development Systems.....	6
Activating XOPs Using the Igor Extensions Folder .....	6
Installing the XOP Toolkit.....	7
XOP Toolkit Overview .....	8
XOPSupport.....	11
The Sample XOPs.....	11
How Igor Integrates XOPs.....	13
The Basic Structure of an XOP.....	15
Preparing to Write an XOP .....	17
Technical Support .....	17

## Chapter 2 - Guided Tour

Overview .....	19
Activating XOPs .....	19
What We Will Do .....	20
Installation .....	20
Building SimpleFit.....	20
Testing SimpleFit64.....	23
Creating a New Project .....	24
Building the New Project in Xcode .....	25
Building the New Project In Visual C++.....	27
Changing the Help .....	28
Changing and Compiling the Code.....	29
Testing SimpleGaussFit.....	30
Touring the Code .....	32
Where To Go From Here .....	34

## Chapter 3 - Development Systems

Overview .....	35
XOPs in Xcode .....	35
XOP Projects in Xcode .....	36

Xcode Project Settings.....	38
Updating Xcode Projects for Xcode 10 .....	44
Debugging an XOP in Xcode .....	46
XOPs in Visual C++ .....	47
Visual C++ Compatibility Issues.....	47
XOP Projects in Visual C++ .....	48
Visual C++ Project Settings.....	49
Debugging a Visual C++ XOP .....	51

## Chapter 4 - Igor/XOP Interactions

Overview.....	53
XOP Resources .....	55
How Igor Calls an XOP .....	57
Basic XOP Messages .....	57
Messages for XOPs that Save and Load Settings .....	59
XOP Errors .....	61
XOPs and Preferences .....	65
XOPs and Experiments.....	65
The IORecHandle .....	66
Messages, Arguments and Results.....	67
Handling Recursion .....	67
Data Sharing .....	69
Igor/XOP Compatibility Issues.....	69

## Chapter 5 - Adding Operations

Overview.....	71
The XOPC 1100 Resource.....	72
Choose a Distinctive Operation Name.....	74
Operation Handler.....	74
Creating Starter Code.....	74
Operation Parameters.....	75
The Command Template .....	76
Optional Parameters.....	77
Mnemonic Names .....	79
The Runtime Parameter Structure.....	79
Qualified Keywords .....	82
String Parameters .....	83
Name Parameters .....	84
Double Name Parameters .....	84
Wave Parameters .....	84
Wave Range Parameters .....	84

VarName Parameters .....	85
Data Folder Reference Parameters.....	86
DataFolderAndName Parameters .....	86
Structure Parameters .....	89
Runtime Output Variables .....	94
Starter Code Details .....	94
Updating Starter Code .....	95
Thread-Safe External Operations.....	95
External Operations and Macintosh Resources .....	97
Operation Handler Checklist .....	98

## Chapter 6 - Adding Functions

Overview.....	99
External Function Examples .....	99
Adding an External Function to Igor .....	100
The XOPF 1100 Resource .....	100
Choose a Distinctive Function Name .....	102
Invoking an External Function .....	103
External Function Parameters and Results .....	103
Parameter and Result Types .....	104
Wave Parameters .....	105
Wave Results .....	106
Data Folder Reference Parameters.....	106
Data Folder Reference Results .....	107
Data Folder Reference Example .....	107
Complex Parameters and Results .....	108
Strings Parameters and Results.....	108
Structure Parameters .....	111
Pass-By-Reference Parameters .....	114
Keep External Functions in Memory .....	116
FUNCTION Message Versus Direct Methods .....	116
Thread-Safe External Functions .....	117
External Functions and Macintosh Resources .....	118
Error Checking and Reporting.....	118

## Chapter 7 - Accessing Igor Data

Overview.....	121
Waves.....	121
Numeric and String Variables.....	138
Dealing With Data Folders .....	141

## **Chapter 8 - Adding Menus and Menu Items**

Overview .....	145
XOPSupport Menu Routines .....	145
Adding Menus and Menu Items.....	147
Responding to Menu Selections .....	150
Enabling and Disabling Menu Items .....	152

## **Chapter 9 - Adding Windows**

Overview .....	155
Guided Tour of WindowXOP1 .....	156
The Architecture of XOP Windows.....	159
Source Code Organization .....	161
Communication With the Operating System .....	162
Communication With Igor .....	162
XOPWindowProxy Class and WindowXOP1Window Class .....	163
XOPContainerProxy Class and WindowXOP1Container Class .....	163
Platform-Independent Source Files .....	163
Macintosh XOP Window Programming.....	165
Macintosh Source Files.....	165
Windows XOP Window Programming .....	166
Window Source Files.....	166
Menu Handling for XOP Windows and Containers .....	166
Cursor Handling for XOP Windows and Containers.....	167
Activating, Showing and Hiding Windows .....	168
Messages for XOP Windows.....	168
Messages for XOP Containers.....	169
Igor Window Coordinates.....	169
Adding XOP Target Windows.....	170
XOP Window Issues .....	171
XOP Subwindow Issues.....	171
Changes to XOP Window Programming in Igor Pro 7 .....	172
TU ("Text Utility") Windows .....	174
Recursion Problems .....	174

## **Chapter 10 - Multithreading**

Multithreading With XOPs .....	175
Thread-safe External Operations and External Functions .....	175
Three Kinds of Threads .....	176
Two Kinds of Igor Preemptive Threads .....	176
Igor Messages and Threads.....	176
Writing Thread-safe Code .....	176

Igor Preemptive Thread Data Environment.....	177
Waves Passed to Igor Threads as Parameters .....	177
Categories of XOPSupport Routines .....	178
Igor-Thread-Safe Callbacks .....	178
Thread-Safety of Callbacks from an Igor Preemptive Thread.....	178
You Can Not Do Callbacks From Private Threads.....	179
Determining if You are Running in a Preemptive Thread.....	180
Strategies For Threaded Data Acquisition.....	180

## Chapter 11 - 64-bit XOPs

Overview .....	187
Activating a 64-bit XOP .....	188
Updating Your Existing Project for 64-bit Development.....	188
Data Types For 64-bit Compatibility .....	188
Programming For 64-bit Compatibility .....	189

## Chapter 12 - Other Programming Topics

The XOPMain Function.....	195
WM Memory XOPSupport Routines .....	195
Techniques for Cross-Platform Development .....	201
Text Encodings .....	201
Text Encodings in Source Files .....	202
File I/O .....	204
File Path Conversions .....	205
Long Names and Paths in XOPs.....	206
Alerts and Message Boxes .....	206
Adding Dialogs .....	207
Adding Version Resources .....	207
Structure Alignment.....	208
Using Igor Structures as Parameters .....	209
Calling User-Defined and External Functions.....	212
Checking For Aborts.....	214
64-bit Integer Issues.....	215
Writing XOPs in C++ .....	216
Windows DLL Dependencies.....	217

## Chapter 13 - Providing Help

Overview.....	219
Igor Pro Help File .....	219
Help for External Operations and Functions .....	219
Programmatically Displaying a Help Topic .....	220

## Chapter 14 - Debugging

Overview .....	221
Programming Problems .....	221
Debugging Techniques .....	232
Avoiding Common Pitfalls .....	233

## Chapter 15 - XOPSupport Routines

About XOPSupport Routines.....	238
Routines for Communicating with Igor .....	239
Operation Handler Routines .....	242
Routines for Accessing Waves .....	253
Routines for Accessing Variables .....	291
Routines for Accessing Data Folders.....	300
Routines for XOPs with Menu Items.....	316
Routines for XOPs that Have Dialogs .....	327
Routines for XOPs that Access Files .....	328
Routines for File-Loader XOPs .....	345
Routines for XOPs with Windows.....	349
Routines for XOPs with Text Windows .....	350
Routines for Dealing with Resources .....	351
Routines for XOPs That Use FIFOs .....	353
Numeric Conversion Routines.....	354
Routines for Dealing With Object Names .....	358
Color Table Routines .....	364
Command Window .....	366
Routines for Dealing With Igor Procedures .....	370
Multithreading Support Routines.....	379
Windows-Specific Routines .....	382
Miscellaneous Routines .....	384
Programming Utilities.....	399
WM Memory Routines .....	410

## Appendix A - XOP Toolkit 7 Upgrade Notes

Overview .....	415
Development Systems Supported by XOP Toolkit 7 .....	415
Create a Copy of Your Existing XOPs .....	415
Changes You Must Make .....	416
Removed Sample XOPs.....	417
Changes from GBLoadWave to GBLoadWaveX.....	417
Memory Management Changes .....	417
Change to Wave Handles.....	417



Changes to Igor Behavior .....	418
New Features .....	418
XOPStandardHeaders Changes .....	418
Organizational Changes .....	418
Text Encodings Used By XOPSupport Routines .....	418
Removed XOPSupport Routines .....	419
Changed XOPSupport Routines .....	419
New XOPSupport Routines .....	420
Fixes for Compiler Changes .....	421
Checking the Igor Version .....	421
XOP Menus in XOP Toolkit 7 .....	421
Help Resources in XOP Toolkit 7 .....	423
XOP Dialogs in XOP Toolkit 7 .....	423
XOP Windows in Igor Pro 7 .....	424
TU ("Text Utility") Windows in Igor Pro 7 .....	424

## **Appendix B - Project Changes For XOP Toolkit 7**

Overview .....	427
Xcode Project Changes For XOP Toolkit 7 .....	427
Converting Your Older Xcode Project for Xcode 9 .....	427
Converting Your Older Xcode Project for Xcode 10 .....	431
Visual C++ Project Changes For XOP Toolkit 7 .....	434
Converting Your Older Visual C++ Project .....	434
Text Encoding Issues .....	439

## **Appendix C - XOP Toolkit 7 Release History**

Overview .....	441
Release 7.00 .....	441
Release 7.01 .....	441
Release 7.02 .....	443
Release 7.03 .....	443

## **Index**



# Chapter 1

## Introduction to XOPs

### About This Manual

This manual describes XOP Toolkit 7.03. It focuses on development using Xcode 7 through Xcode 10 on Macintosh and Visual C++ 2015 or Visual C++ 2017 on Windows. XOP Toolkit 7 creates XOPs that run with Igor Pro 6.20 or later.

If you are targeting Igor Pro 8.00 or later, and you are creating a new XOP or updating an XOP already updated using XOP Toolkit 7.01 or later, you should use XOP Toolkit 8. See **Choosing Which XOP Toolkit to Use** on page 5 for details.

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 6, we recommend that you read this chapter and then **XOP Toolkit 7 Upgrade Notes** on page 415. As explained there, we recommend that you leave your existing XOP project folders as they are and create new copies of those folders for further development using XOP Toolkit 7.

For a list of changes to XOP Toolkit 7 since release 7.00, see **XOP Toolkit 7 Release History** on page 441.

### 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, and error messages.

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 MacOS. It was based on Apple's Carbon API which supported software running on both Mac OS 9 and MacOS. 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 MacOS, 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 MacOS. 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 MacOS 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 MacOS 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 MacOS 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.

In February of 2013, WaveMetrics released Igor 6.30 which ran on MacOS 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.

In July of 2016, WaveMetrics released Igor Pro 7.00. Igor Pro 7 is a major overhaul of Igor Pro 6 and is built on the Qt cross-platform framework. Igor Pro 7 is available in both 32 bits and 64 bits and is the first version of Igor that runs in 64 bits on Macintosh.

Also in July of 2016, WaveMetrics released XOP Toolkit 7.00. XOP Toolkit 7.00 was the first toolkit that supported the creation of 64-bit Macintosh XOPs. XOPs created with XOP Toolkit 7 require Igor Pro 6.20 or later. The cross-platform dialog support of XOP Toolkit 6 was removed. The GBLoadWaveX XOP was added to show how to create a dialog using platform-specific code, replacing the old GBLoadWave XOP. The protocol for creating an XOP window was completely changed. As relates to text encoding, the XOP Toolkit 7 sample projects are optimized for use with Igor7, by virtue of using UTF-8 for source files.

Most pre-existing XOPs work with Igor7. XOPs that add menus or dialogs may need to be updated using XOP Toolkit 7. XOPs that add windows must be updated using XOP Toolkit 7. To create a 64-bit Macintosh XOP, you must use XOP Toolkit 7. See **XOP Toolkit 7 Upgrade Notes** on page 415 for details on updating an existing XOP for XOP Toolkit 7.

In April of 2017, WaveMetrics released XOP Toolkit 7.01 which was tested in Xcode 7 and 8 and Visual C++ 2015 and 2017. XOP Toolkit 7.01 introduced new memory management routines described under **WM Memory XOPSupport Routines** on page 195. Use of these routines is required for compatibility with the 64-bit version of Igor Pro 8.

In May of 2018, WaveMetrics released Igor Pro 8.00. Igor8 supports object names up to 255 bytes in length. Object names in earlier versions of Igor were limited to 31 bytes. Igor8 supports old XOPs compiled with the 31-byte object name limit using XOP Toolkit 7 or before, but generates an error if passing a long object name to the XOP is required. On Macintosh, Igor8 is available only in the 64-bit version, not in 32 bits. Consequently, existing 32-bit Macintosh XOPs can not run with Igor8. On Windows, Igor8 is available in both 64-bit and 32-bit versions, but 64 bits is recommended.

In June of 2018, WaveMetrics released XOP Toolkit 8.00 which was tested in Xcode 9 and Visual C++ 2015 and 2017. XOP Toolkit 8.00 supports long object names. Consequently, XOPs compiled using XOP Toolkit 8 require Igor Pro 8.00 or later. See **Choosing Which XOP Toolkit to Use** on page 5 for details.

Support for Xcode 10 was added in XOP Toolkit 7.03 and XOP Toolkit 8.01 in September of 2018. Xcode 10 supports 64-bit development only, not 32-bit development.

## XOP Toolkit 7

XOPs created with XOP Toolkit 7 require Igor Pro 6.20 or later. Some features require later versions of Igor. This is noted in the documentation where applicable.

The section **Igor/XOP Compatibility Issues** on page 69 explains how you can check which version of Igor you are running with.

In most cases you can easily support Igor6 (6.20 or later) and Igor7 with one XOP. An exception is if you make frequent use of non-ASCII text. In that case, you may be better off maintaining separate Igor6 and Igor7 versions of your XOP. See **Text Encodings** on page 201 for details.

## Choosing Which XOP Toolkit to Use

As of this writing (2018), WaveMetrics is shipping two versions of the XOP Toolkit - version 7 and version 8.

Use XOP Toolkit 7 if your XOP must run with Igor Pro 6 or Igor Pro 7, or if you are updating an older XOP that was never updated using XOP Toolkit 7.01 or later, or if you have not yet created a 64-bit version of your XOP.

Use XOP Toolkit 8 if you are writing a new XOP to target Igor Pro 8 or later, or if you are updating an XOP that was already updated using XOP Toolkit 7.01 or later to target Igor Pro 8 or later.

The next section explains the reason for these rules.

### Main Issues

There are three main issues:

#### 1. 64-bit XOPs

If you have not ported your existing XOP to 64 bits, use XOP Toolkit 7 to do that. Chapter 11 of the XOP Toolkit 7 manual includes detailed instructions for porting to 64 bits. To reduce clutter, the detailed instructions were removed from the XOP Toolkit 8 manual.

#### 2. Memory Management Routines

To support blocks of memory larger than 2 GB, WaveMetrics changed Igor's internal memory management routines for Igor Pro 8. No change is required for 32-bit Macintosh XOPs or for 32-bit or 64-bit Windows XOPs. However, to run with Igor Pro 8 or later, 64-bit Macintosh XOPs must be modified as explained under "WMMemory XOPSupport Routines" in Chapter 12 of the XOP Toolkit 7 manual.

Although it is not required to update 32-bit Macintosh XOPs and 32-bit or 64-bit Windows XOPs, we recommend that you do so to keep your code up-to-date.

#### 3. Long Object Names and Paths

Prior to Igor Pro 8, object names, such as the names of waves, variables, and data folders, were limited to 31 bytes. In Igor Pro 8.00, this limit was raised to 255 bytes, and other limits were also raised. For compatibility with older XOPs, Igor Pro 8 and later can run with XOPs that do not support long names and paths, but return errors if passing a long name or path to the XOP is required.

If your XOP uses any of the following constants then it uses object names and/or paths:

MAX\_OBJ\_NAME, MAX\_WAVE\_NAME, MAX\_DIM\_LABEL\_BYTES  
MAXCMDLEN, MAX\_PATH\_LEN

XOP Toolkit 7 supports short names and paths only. It can create XOPs that run with Igor Pro 6.20 or later, but they will return errors if used with long names or paths in Igor Pro 8.00 or later.

XOP Toolkit 8 supports long names and paths only. Long names and paths require Igor Pro 8.00 or later. Consequently, you should use XOP Toolkit 8 only if you are willing to require Igor Pro 8.00 or later. See "Long Names and Paths in XOPs" in Chapter 12 of the XOP Toolkit 8 manual for details.

### Summary of Recommendations

If you are updating an XOP that was never updated using XOP Toolkit 7.01 or later, or if you have not yet created a 64-bit version of your XOP, first update your XOP using XOP Toolkit 7. See Appendix A in the XOP Toolkit 7 manual for details.

Once you have updated your XOP with XOP Toolkit 7, if you want to support long object names and paths and are willing to require Igor Pro 8.00 or later, update using XOP Toolkit 8. See Appendix A in the XOP Toolkit 8 manual for details.

## Chapter 1 — Introduction to XOPs

---

Here is a summary of the rules for determining which XOP Toolkit to use:

- If you are creating a new XOP and you want to run with Igor Pro 8.00 or later, use XOP Toolkit 8.
- If you are creating a new XOP and you want to run with Igor Pro 6.20 or later, use XOP Toolkit 7.
- If you are updating an XOP created with XOP Toolkit 7.00 or before, or if you have not yet created a 64-bit version of your XOP, first update using XOP Toolkit 7. Then, if you want to run with Igor Pro 8.00 or later, update using XOP Toolkit 8.

## Macintosh and Windows XOPs

XOP Toolkit 7 supports development of 32-bit and 64-bit XOPs for both Macintosh and Windows.

Some XOPs, by their nature, are platform-independent. For example, the code for a number-crunching XOP will be 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 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.

## Development Systems

This table lists the development systems supported by XOP Toolkit 7.

Development System	XOPs Run On	Dev Sys Files In	Notes
Xcode 7 through Xcode 10	MacOS 10.6 or later	<project>/Xcode	
Visual C++ 2015 or 2017	Windows 7 or later	<project>/VC	Any edition is OK.

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

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.

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 (*Macintosh*) or shortcut (*Windows*) pointing to it, in the “Igor Extensions” or “Igor Extensions (64-bit)” folder.

There are Igor Extensions folders in the Igor Pro Folder but you should not use them to activate XOPs. Instead use the Igor Extensions folders in the “Igor Pro User Files” folder. These folders are created by Igor the first time you run the program.

The term “Igor Pro User Files” folder is generic. The actual folder name includes the Igor version, such as “Igor Pro 6 User Files” or “Igor Pro 7 User Files”.



This table shows the default location of the Igor7 Igor Extensions folders on each operating system:

OS	Path
MacOS	/Users/<user>/Documents/WaveMetrics/Igor Pro 7 User Files/Igor Extensions
	/Users/<user>/Documents/WaveMetrics/Igor Pro 7 User Files/Igor Extensions (64-bit)
Windows	C:\Users\<user>\Documents\WaveMetrics\Igor Pro 7 User Files\Igor Extensions
	C:\Users\<user>\Documents\WaveMetrics\Igor Pro 7 User Files\Igor Extensions (64-bit)

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

To activate an XOP, put the XOP, or an alias or shortcut pointing to it, in the “Igor Extensions” or “Igor Extensions (64-bit)” folder inside the “Igor Pro User Files” folder.

If you are activating a WaveMetrics XOP that resides in “Igor Pro Folder/More Extensions” or “Igor Pro Folder/More Extensions (64-bit)”, make an alias or shortcut for the XOP and put the alias or shortcut in “Igor Pro User Files/Igor Extensions” or “Igor Pro User Files/Igor Extensions (64-bit)”.

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 “Igor Pro User Files/Igor Extensions (64-bit)”, 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 7” 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 XOPToolkit7.dmg (*Macintosh*) or XOPToolkit7.exe (*Windows*) and install the “XOP Toolkit 7” folder on your hard disk.

All of the sample XOPs, the XOPSupport library and other support files are stored in a folder named IgorXOPs7 inside the “XOP Toolkit 7” folder.

## Testing the Macintosh Installation

These instructions apply to Xcode 7 through Xcode 10.

The instructions assume that you are running Igor7 or later and therefore show how to build and activate the 64-bit version of the XOP and run it with the 64-bit version of Igor. If you are running Igor6, build and activate the 32-bit version and run it with the 32-bit version of Igor.

1. In the Finder, open the IgorXOPs7/XFUNC1/Xcode folder.
2. Open the XFUNC1.xcodeproj project in Xcode.  
If you are running Xcode 10 or later, Xcode will display a dialog saying “32-bit MacOS targets are no longer supported”. This is because Xcode 10 no longer supports 32-bit development. Click OK to dismiss the dialog.
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.

## Chapter 1 — Introduction to XOPs

---

7. Click Done to close the Project Settings dialog.
8. Choose Product→Scheme→XFUNC1-64.
9. Choose Product→Build.

This creates the XFUNC1-64.xop package in the IgorXOPs7\XFUNC1\Xcode\build\Debug. Don't worry if you got some warnings from the compiler during the build.

10. Make an alias from the XFUNC1-64.xop package and drag the alias into your "Igor Extensions (x64)" folder.
11. Launch the 64-bit version of Igor.
12. Execute the following command:

```
Print XFUNC1Add(3,4)
```

13. Igor should print "7" in the history area.

If Igor displayed an error message, make sure that you activated XFUNC1-64.xop correctly. See **Activating XOPs Using the Igor Extensions Folder** on page 6 for details. Also make sure that you ran the 64-bit version of Igor.

### Testing the Windows Installation

These instructions apply to Visual C++ 2015. They were also tested with Visual C++ 2017. You may get compile or link errors with another version.

The instructions assume that you are running Igor7 or later and therefore show how to build and activate the 64-bit version of the XOP and run it with the 64-bit version of Igor. If you are running Igor6, build and activate the 32-bit version and run it with the 32-bit version of Igor.

1. In the Windows desktop, open the IgorXOPs7\XFUNC1\VC folder.
2. Open the XFUNC1.sln file in Visual C++.
3. From the platform popup menu in the standard toolbar (under the menu bar), choose x64.
4. Choose Build Solution from the Build menu. This creates the XFUNC1-64.xop file in the IgorXOPs7\XFUNC1\VC folder. Don't worry if you got some warnings from the compiler during the build.
5. Make a shortcut from the XFUNC1-64.xop file and drag the shortcut into your "Igor Extensions (x64)" folder.
6. Launch the 64-bit version of Igor.
7. Execute the following command:  

```
Print XFUNC1Add(3,4)
```
8. Igor should print "7" in the history area.

If Igor displayed an error message, make sure that you activated XFUNC1-64.xop correctly. See **Activating XOPs Using the Igor Extensions Folder** on page 6 for details. Also make sure that you ran the 64-bit version of Igor.

### 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 IgorXOPs7 folder.

The IgorXOPs7 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 memory management 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.
/Xcode/libXOPSupport.a	The Xcode 6 XOPSupport libraries for MacOS XOPs. Requires MacOS 10.6 or later. Works with Xcode 7 and 8.
/Xcode/libXOPSupport64.a	
\VC\XOPSupport.lib	The Visual C++ 2015 XOPSupport libraries for Windows XOPs.
\VC\XOPSupport64.lib	
IGOR.lib	Allows a Windows XOP to call memory management and other utility routines that are implemented in the Igor.exe file on Windows.
IGOR64.lib	

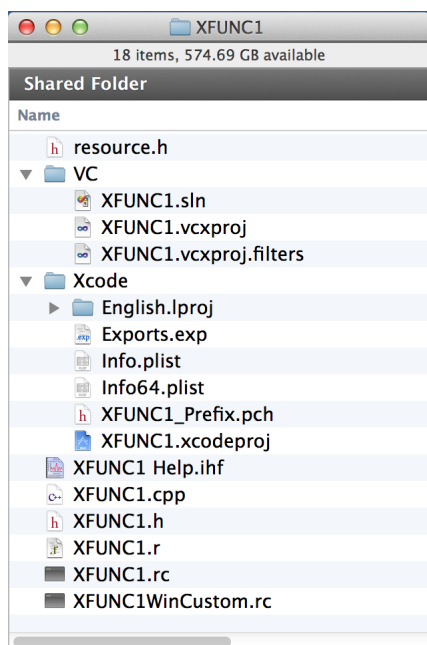
libXOPSupport.a, XOPSupport.lib and IGOR.lib are used when building 32-bit XOPs only.

libXOPSupport64.a, 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.	XFUNG1.cpp
A header file containing #defines and other declarations.	XFUNG1.h
A resource file containing a text description of the XOP's resources.	XFUNG1.r ( <i>Macintosh</i> ) or XFUNG1.rc ( <i>Windows</i> )
A "custom" resource file containing a text description of XOP-specific resources ( <i>Windows</i> only)	XFUNG1WinCustom.rc
A resource header file containing defines used in the main resource file ( <i>Windows</i> only)	resource.h
A help file. This is an Igor Pro notebook that has been formatted and opened in Igor Pro as a help file.	XFUNG1 Help.ihf
A "project", and/or "solution" file.	XFUNG1.xcodeproj (Xcode)
In Visual C++, open the solution file (.sln) to work with the project.	XFUNG1.sln (VC) XFUNG1.vcxproj (VC) XFUNG1.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 (Xcode or VC).

## 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 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). 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.

### 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.

### GBLoadWaveX

GBLoadWaveX loads general binary files into Igor waves.

GBLoadWaveX is the descendent of the GBLoadWave XOP that was provided as a sample XOP in XOP Toolkit 1 through XOP Toolkit 6. In Igor7, GBLoadWave was built into Igor itself, so, in XOP Toolkit 7, the sample XOP was renamed GBLoadWaveX. See **Changes from GBLoadWave to GBLoadWaveX** on page 417 for a discussion of differences between GBLoadWaveX and GBLoadWave.

The GBLoadWaveX XOP adds the GBLoadWaveX command line operation to Igor and is a good starting point for XOPs that import binary data. It also illustrates how to create an Igor-style dialog using platform-dependent code.

### SimpleFit

SimpleFit adds a simple curve-fitting function for fitting 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.

### 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.

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

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 69.)

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.

## Chapter 1 — Introduction to XOPs

---

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 the history area of Igor’s command window. There are also callbacks that allow an XOP to execute a standard Igor command.

A simple XOP may use just a few callbacks.



## 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

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`
- 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

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 69 and **WM Memory XOPSupport Routines** on page 195 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 61.

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 207.

If your XOP saves its settings or documents as part of an Igor experiment, read **XOPs and Experiments** on page 65.

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 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

### Email Support

You can send questions to us via email at [support@wavemetrics.com](mailto:support@wavemetrics.com).

For sales matters, please use [sales@wavemetrics.com](mailto:sales@wavemetrics.com).

### World-Wide Web

Our Web address is [www.wavemetrics.com](http://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 7.03 of the Igor XOP Toolkit which supports development using Xcode 7 through Xcode 10 on Macintosh and Visual C++ 2015 and Visual C++ 2017 on Windows.

XOP Toolkit 7.03 creates XOPs that run with Igor Pro 6.20 or later.

If you are targeting Igor Pro 8.00 or later, and you are creating a new XOP or updating an XOP already updated using XOP Toolkit 7.01 or later, you should use XOP Toolkit 8. See **Choosing Which XOP Toolkit to Use** on page 5 for details.

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 7 through Xcode 10 on MacOS 10.9 or later
- Microsoft Visual C++ 2015 or 2017 (including the free “Community” edition)

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

#### 1-Xcode.

<Step for Xcode 7 through Xcode 10>.

#### 1-Visual C++.

<Step for Visual C++ 2015 or 2017>.

As explained under **Visual C++ Compatibility Issues** on page 47, you may get compile or link errors with another version of Visual C++.

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

This guided tour assumes that you are using the 64-bit version of Igor7 or later. The instructions will work with the 32-bit version of Igor6 or Igor7 if you remove “64” from file and folder names.

### Activating XOPs

In the guided tour, you will be instructed to activate an XOP multiple times. To activate a 64-bit XOP put an alias or shortcut pointing to the XOP in the “Igor Extensions (64-bit)” folder inside “Igor Pro 7 User Files” folder. For details, see **Activating XOPs Using the Igor Extensions Folder** on page 6.

### 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 IgorXOPs7 folder and contains the following project files:

Xcode/SimpleFit.xcodeproj	For Xcode 7 through Xcode 10. The compiled XOP will run on MacOS 10.6 or later.
VC\SimpleFit.sln	For Visual C++ 2015 or 2017 on Windows. The compiled XOP will run on Windows 7 or later.

---

### Building SimpleFit in Xcode

These steps work in Xcode 7 through Xcode 10.

#### 1-Xcode.

**Open the SimpleFit/Xcode/SimpleFit.xcodeproj project file in Xcode.**

The SimpleFit project window is shown.

If you are using Xcode 10 or later, Xcode will display a dialog saying “32-bit MacOS targets are no longer supported”. Click OK.

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.

## 2-Xcode

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

Shared Project Settings:

Build System: Standard Build System (Default)

Per-User Project Settings:

Build System: Use Shared Setting

Derived Data: Default Location

/Users/Howard/Libr...r/Xcode/DerivedData

Advanced...

Issues: ☒ Show live issues for source code  
☒ Show issues for active scheme only  
☐ Show all issues

Done

Although this dialog is called “Project Settings”, Xcode 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:

Build Location

☐ Xcode Default  
Use the location specified in Xcode preferences

☐ Unique  
In a unique subfolder of Xcode's Derived Data location

☐ Shared Folder  
In a shared subfolder of Xcode's Derived Data location

☒ Custom  
Relative to Derived Data  
Fully specify locations of build products and intermediates

Products

Intermediates

Index Datastore

☒ Legacy  
Use target settings (used for legacy projects that cannot be relocated)

Products: Specified per target

Intermediates: Specified per target

Index Datastore: ~/Library/Developer/Xcode/Derive...kydcbrrnmvmjoozy/Index/DataStore

☐ Full Index Datastore Visibility  
Include index data information for files not referenced from the workspace

Done

Click the **Legacy** radio button.

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

## Chapter 2 — Guided Tour

Click Done to close the Advanced Project Settings dialog.

Click Done to close the Project Settings dialog.

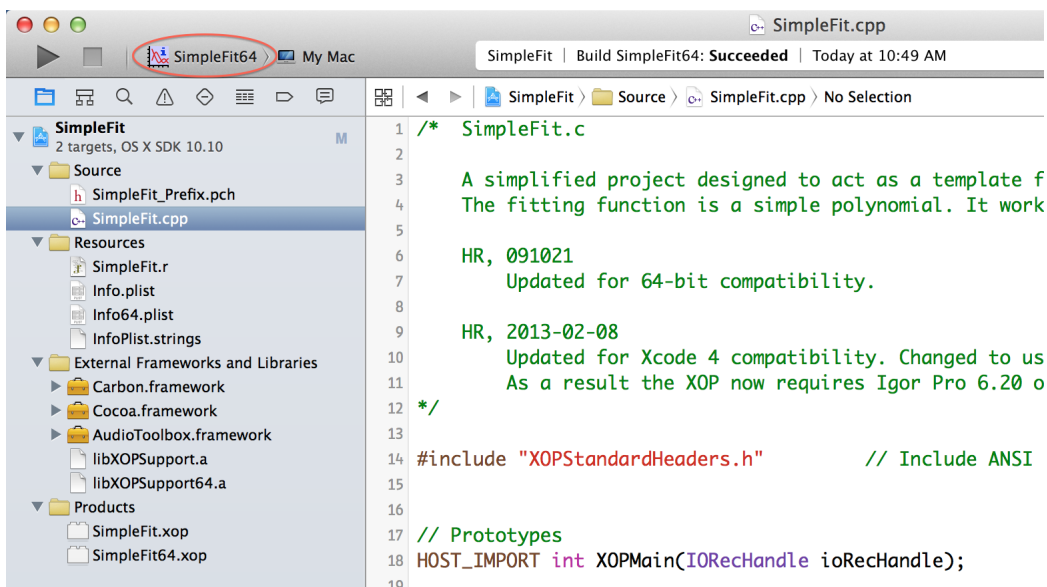
### 3-Xcode.

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

### 4-Xcode.

Choose Product→Scheme→SimpleFit64.

The project window should now look like this:



### 5-Xcode.

Choose Product→Build.

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

In a few seconds the compiler will finish and an XOP package named SimpleFit64.xop will be created in your SimpleFit/Xcode/build/Debug folder. Note where this is located for use in a upcoming step.

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

### 6-Xcode.

Close the project window.

## Building SimpleFit in Visual C++

### 1-Visual C++.

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

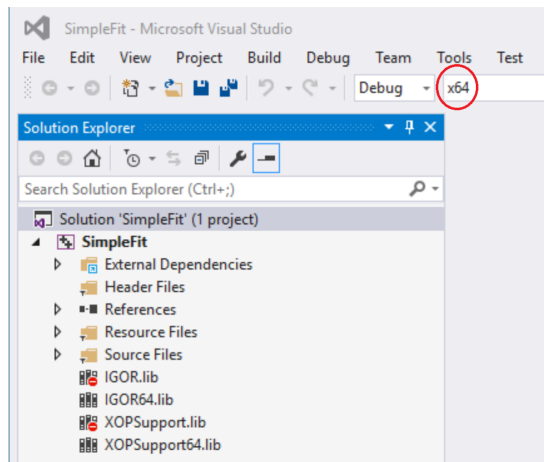
The SimpleFit project window is shown.

Choose View→Solution Explorer and open the SimpleFit icon.

From the Solution Platforms popup menu, choose x64.

The window should now look like this:





## 2-Visual C++.

**Choose Build→Build Solution.**

In a few seconds the compiler will finish and an XOP file named SimpleFit64.xop will be created in your SimpleFit\VC 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++.

**Choose File→Close Solution.**

# Testing SimpleFit64

Now that we have built the SimpleFit64 XOP, our next task is to test it.

In the next step, we activate the SimpleFit64 XOP so that Igor will load it. For a detailed explanation of activating, see **Activating XOPs Using the Igor Extensions Folder** on page 6.

1. **On the desktop, make an alias (Macintosh) or shortcut (Windows) for the newly created SimpleFit64.xop and drag the alias or shortcut into the "Igor Extensions (64-bit)" 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 (SimpleFit64.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 the 64-bit version of Igor Pro.**

The 64-bit version is named Igor64.app on Macintosh, Igor64.exe on Windows.

You must restart Igor Pro after changing the contents of the "Igor Extensions (64-bit)" folder.

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

The Igor Help Browser appears with the Command Help tab selected.

5. **Find SimpleFit in the list of topics and select it.**

The custom help for the SimpleFit external function is shown. This help is stored in the "SimpleFit Help.ihf" file.

6. **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.

### 7. 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 IgorXOPs7 folder, to clone a project.

#### 1. On the desktop, locate your IgorXOPs7 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.

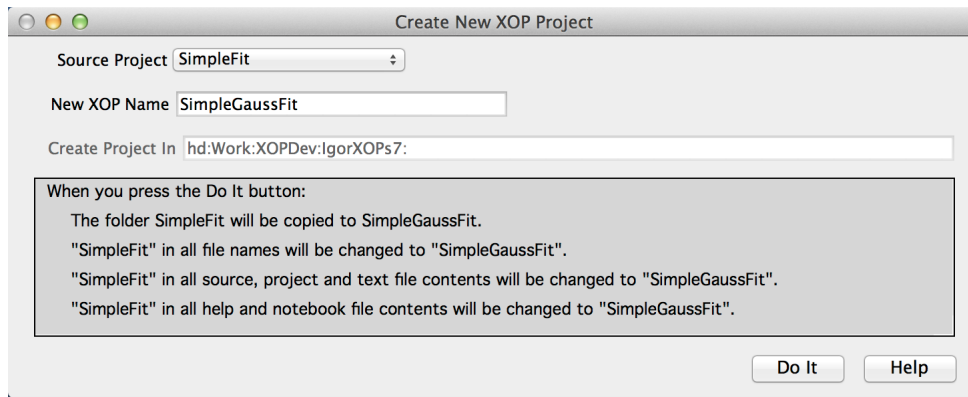
#### 4. In the New XOP Name textbox, enter SimpleGaussFit.

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 IgorXOPs7 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

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

**1-Xcode.**

**On the desktop, open the Xcode folder in the SimpleGaussFit folder.**

This folder contains files specific to the Xcode development system.

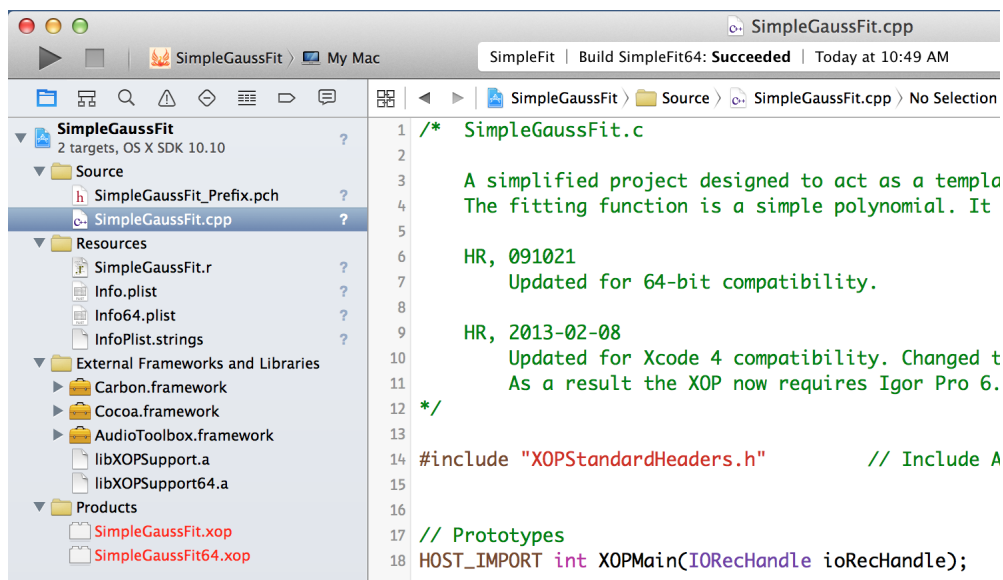
**2-Xcode.**

**Open the SimpleGaussFit.xcodeproj file in Xcode.**

If you are using Xcode 10 or later, Xcode will display a dialog saying “32-bit MacOS targets are no longer supported”. Click OK.

This is what the project window looks like in Xcode 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.

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.

### 4-Xcode.

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

### 5-Xcode.

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.

Click Done to close the Advanced Project Settings dialog.

### 7-Xcode.

Click Done to close the Project Settings dialog.

### 8-Xcode.

Choose Product→Scheme→SimpleGaussFit64.

### 9-Xcode.

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.

### 10-Xcode.

On the desktop, verify that you have a package folder named SimpleGaussFit64.xop in IgorXOPs7/SimpleGaussFit/Xcode/build/Debug.

Debug is the name of the Xcode configuration currently being built so the output goes into the

/build/Debug directory.

The SimpleGaussFit64.xop package folder is your compiled 64-bit 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 SimpleGaussFit64. This is the executable file.

### 11-Xcode.

**Make an alias for the SimpleGaussFit64.xop package folder and put the alias in your “Igor Extensions (64-bit)” folder.**

We now have a SimpleGaussFit64 XOP but it behaves just like the SimpleFit64 XOP. We will fix that in the following sections.

## Building the New Project In Visual C++

### 1-Visual C++.

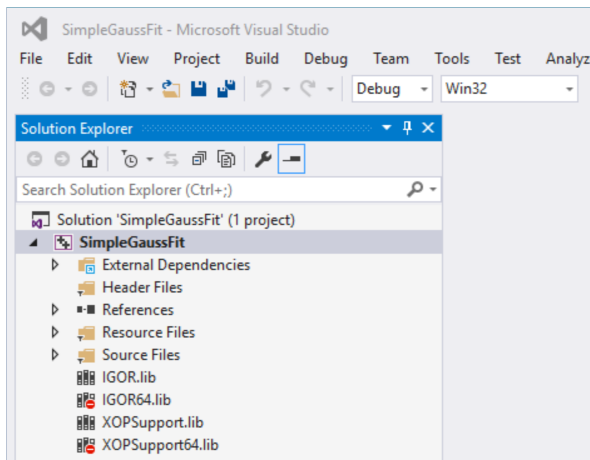
**On the desktop, open the VC folder in the SimpleGaussFit folder.**

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

### 2-Visual C++.

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

The SimpleGaussFit project window is shown:



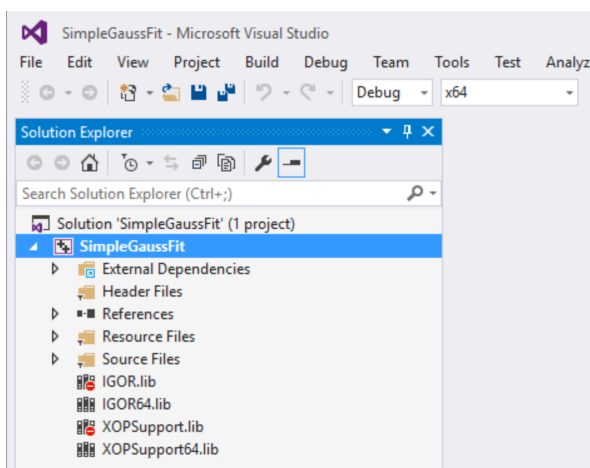
### 3-Visual C++.

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

### 4-Visual C++.

**From the Solution Platforms popup menu, choose x64.**

The SimpleGaussFit project window now looks like this:



### 5-Visual C++.

#### **Choose Build→Build Solution.**

Visual C++ should build the XOP with no errors (though you may get some warnings), creating the file SimpleGaussFit64.xop in your IgorXOPs7\SimpleGaussFit\VC 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.

### 6-Visual C++.

#### **On the desktop, verify that you have a file named SimpleGaussFit64.xop in IgorXOPs7/SimpleGaussFit/VC.**

This is the compiled XOP file.

### 7-Visual C++.

#### **Make a shortcut for the SimpleGaussFit64.xop file and put the shortcut in your “Igor Extensions (64-bit)” folder.**

We now have a SimpleGaussFit64 XOP but it behaves just like the SimpleFit64 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 the 64-bit version of Igor Pro.**
2. **Choose File→Open File→Notebook to display the Open File dialog.**  
We are opening the help file as a notebook so we can edit it.
3. **Select All Files from the file type popup menu in the Open File dialog.**
4. **Open the “SimpleGaussFit Help.ihf” file in the SimpleGaussFit folder.**
5. **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.

6. **Save the notebook by choosing File→Save Notebook.**
5. **Close the notebook window. When Igor asks if you want to kill or hide it, click Kill.**  
Now we will compile the help file so that SimpleGaussFit will provide help in Igor’s Help Browser.
8. **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.**
9. **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.
10. **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.

**Choose Product→Scheme→SimpleGaussFit64.**

**Compile the project by choosing Product→Build.**

The compile and link should proceed without errors (although you may get some warnings), creating SimpleGaussFit64.xop in the IgorXOPs7/SimpleGaussFit/Xcode/build/Debug folder. If you get errors, carefully check the changes that you made to the source code.

### 5-Visual C++.

**Choose x64 from the Solution Platforms popup menu.**

**Compile the project by choosing Build→Build Solution.**

The compile and link should proceed without errors (although you may get some warnings), creating SimpleGaussFit64.xop in the IgorXOPs7\SimpleGaussFit\VC 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 (SimpleGaussFit64.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 the 64-bit version of Igor Pro.**

Igor Pro will load the SimpleGaussFit function into memory.

**4. Choose Help→Command Help.****5. 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 SimpleGaussFit64 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 SimpleGaussFit64.xop and placed the alias or shortcut in the “Igor Extensions (64-bit)” folder. See **Activating XOPs Using the Igor Extensions Folder** on page 6 for details.
- Make sure that you are running the 64-bit version of Igor, not the 32-bit version.
- If you have multiple copies of Igor on your machine, make sure that you are running the copy you intended to run by choosing Help→Show Igor Pro Folder.

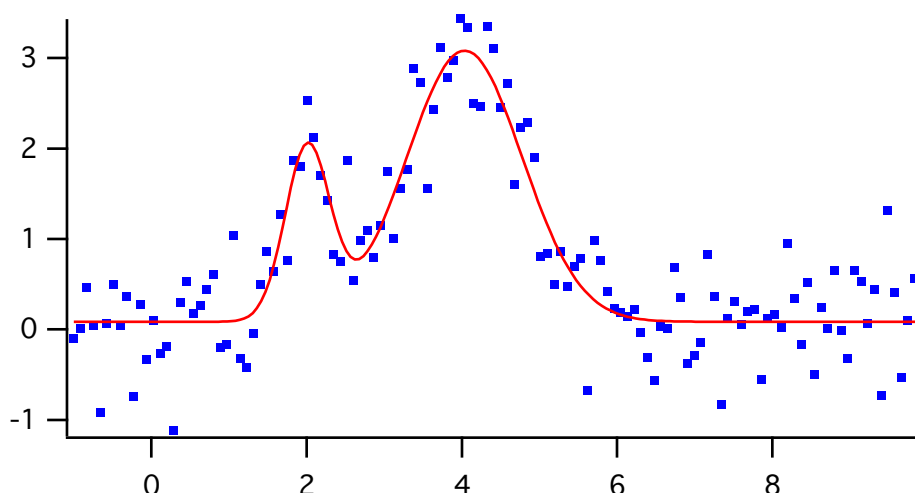
If you do see SimpleGaussFit in the list of functions but get a message saying help is not available when you select it:

- 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 28.

**6. Close the Help Browser.****7. 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.

**8. 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 7 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 57 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.

**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++.

**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"
```

`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 55.

## 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 7 creates XOPs that run with Igor Pro 6.20 or later. It works with Xcode 7 through Xcode 10, Visual C++ 2015, and Visual C++ 2017.

If you are targeting Igor Pro 8.00 or later, and you are creating a new XOP or updating an XOP already updated using XOP Toolkit 7.01 or later, you should use XOP Toolkit 8. See **Choosing Which XOP Toolkit to Use** on page 5 for details.

XOP Toolkit 7 includes sample XOPs and supporting files for the following development systems:

Development System	XOP Runs On	With Igor Version
Xcode 7 through Xcode 10	MacOS 10.6 or later	Igor Pro 6.20 or later
Visual C++ 2015 and 2017	Windows 7 or later	Igor Pro 6.20 or later

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

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

If you use a newer version of Visual C++ on Windows, you may need to recompile the XOPSupport libraries and any other libraries to which you statically link, as explained under **Visual C++ Compatibility Issues** on page 47.

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

On Macintosh, XOP Toolkit 7 supports development using Xcode 7 through Xcode 10. It will likely work with later versions of Xcode.

Xcode 10 supports 64-bit development only, not 32-bit development. Xcode 9 and before support both 32-bit and 64-bit development.

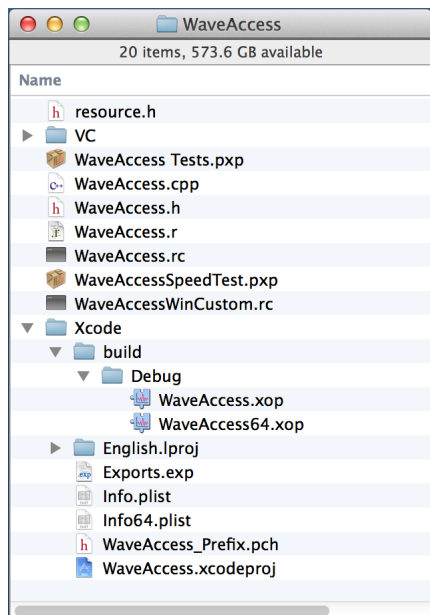
Each sample project contains an Xcode folder. The Xcode folder, e.g., WaveAccess/Xcode, contains the Xcode project file, e.g., WaveAccess.xcodeproj.

For instructions on converting an XOP Toolkit 6 project for use with XOP Toolkit 7, see **Xcode Project Changes For XOP Toolkit 7** on page 427.

### 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 24.

We will use the WaveAccess sample XOP as a case in point. The WaveAccess folder is inside the IgorXOPs7 folder and looks like this:



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

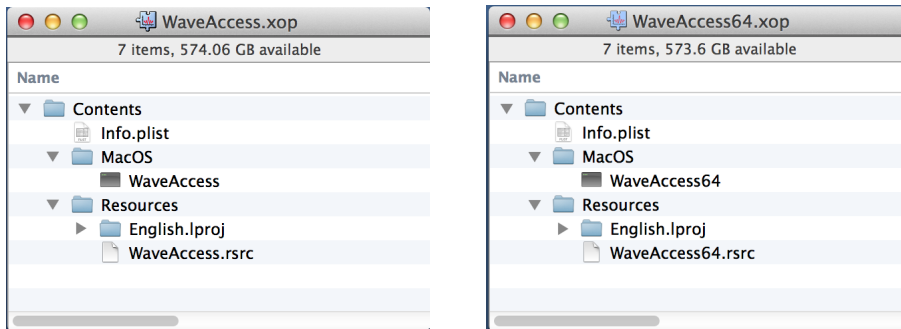
WaveAccess.xcodeproj is the project file and contains the project settings. (Actually it is a package folder, not a file.) The build folder contains data created by the compiler such as object code as well as the compiled XOP. The Xcode project includes two targets: a 32-bit target named WaveAccess and a 64-bit target named WaveAccess64.

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 55. 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.

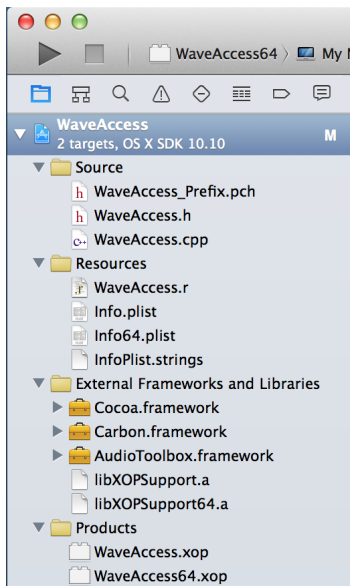
WaveAccess.xop and WaveAccess64.xop appear to be files but are really a "packages". If you control-click and choose Show Package Contents, you see what is inside:



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 32-bit file `WaveAccess` and the 64-bit file `WaveAccess64`. The other files contain data describing the package and resources.

This Xcode 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 `Info64.plist` files store “meta-information” such as version information and the name of the executable file within the package folder.

The `.framework` files makes Apple APIs accessible to the XOP.

`libXOPSupport.a` is the 32-bit XOPSupport library.

`libXOPSupport64.a` is the 64-bit XOPSupport library.

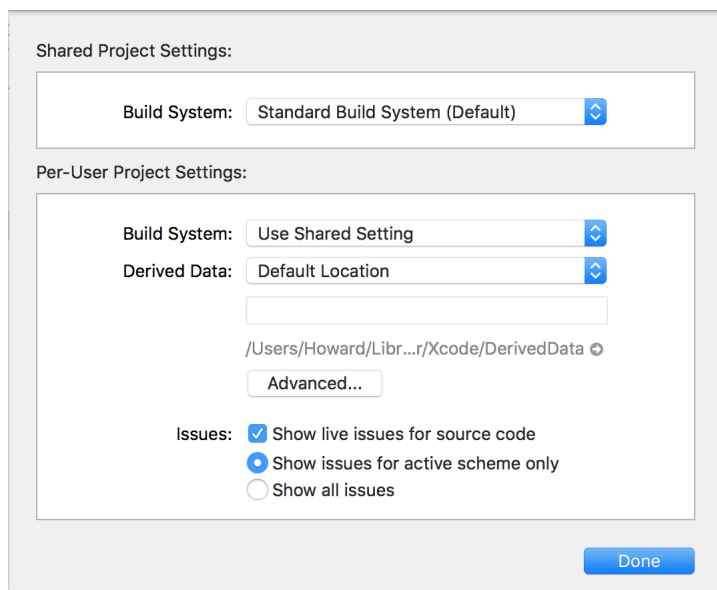
# Xcode Project Settings

In Xcode, 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 Project Settings Dialog

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



The Xcode Project Settings dialog is divided into two sections: "Shared Project Settings" and "Per-User Project Settings".

**Shared Project Settings:**

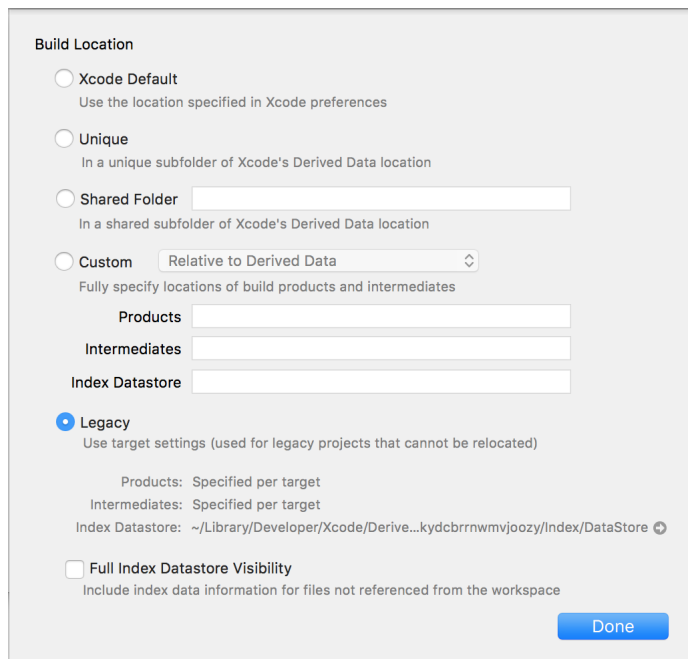
- Build System:** A dropdown menu showing "Standard Build System (Default)".

**Per-User Project Settings:**

- Build System:** A dropdown menu showing "Use Shared Setting".
- Derived Data:** A dropdown menu showing "Default Location". Below it is a text field containing the path "/Users/Howard/Libr...r/Xcode/DerivedData" and a button labeled "Advanced...".
- Issues:** Three radio buttons: "Show live issues for source code" (checked), "Show issues for active scheme only", and "Show all issues".

A "Done" button is located at the bottom right of the dialog.

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



The Build Location subdialog is titled "Build Location" and contains several options:

- Xcode Default:** A radio button with the description "Use the location specified in Xcode preferences".
- Unique:** A radio button with the description "In a unique subfolder of Xcode's Derived Data location".
- Shared Folder:** A radio button with a text field for specifying a folder. The description is "In a shared subfolder of Xcode's Derived Data location".
- Custom:** A radio button with a dropdown menu showing "Relative to Derived Data". The description is "Fully specify locations of build products and intermediates". Below this are three text fields: "Products", "Intermediates", and "Index Datastore".
- Legacy:** A radio button with the description "Use target settings (used for legacy projects that cannot be relocated)". Below this are three text fields: "Products: Specified per target", "Intermediates: Specified per target", and "Index Datastore: ~/Library/Developer/Xcode/Derive...kydcbrnwmvjoozy/Index/DataStore".

There is also a checkbox for "Full Index Datastore Visibility" with the description "Include index data information for files not referenced from the workspace".

A "Done" button is located at the bottom right of the 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>/Xcode/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.

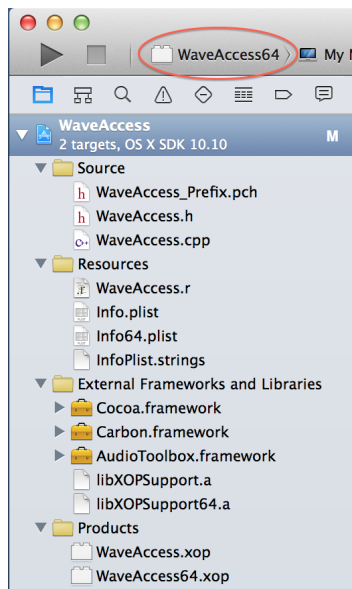
**NOTE:** If you do not set the build location to Legacy your compiled XOP will be created in a strange location such as ~Developer/Xcode/Derived Data. So remember to set this setting for each XOP project you open in Xcode.

**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 Schemes

An Xcode *scheme* is a collection of settings that control things like which target (32-bit or 64-bit) to build, which configuration (debug or release) to build, and which application to run when you ask to debug the program. You choose which scheme to use through the Active Scheme popup menu in the top/left corner of the Xcode main window. Here the scheme name is shown as WaveAccess64:

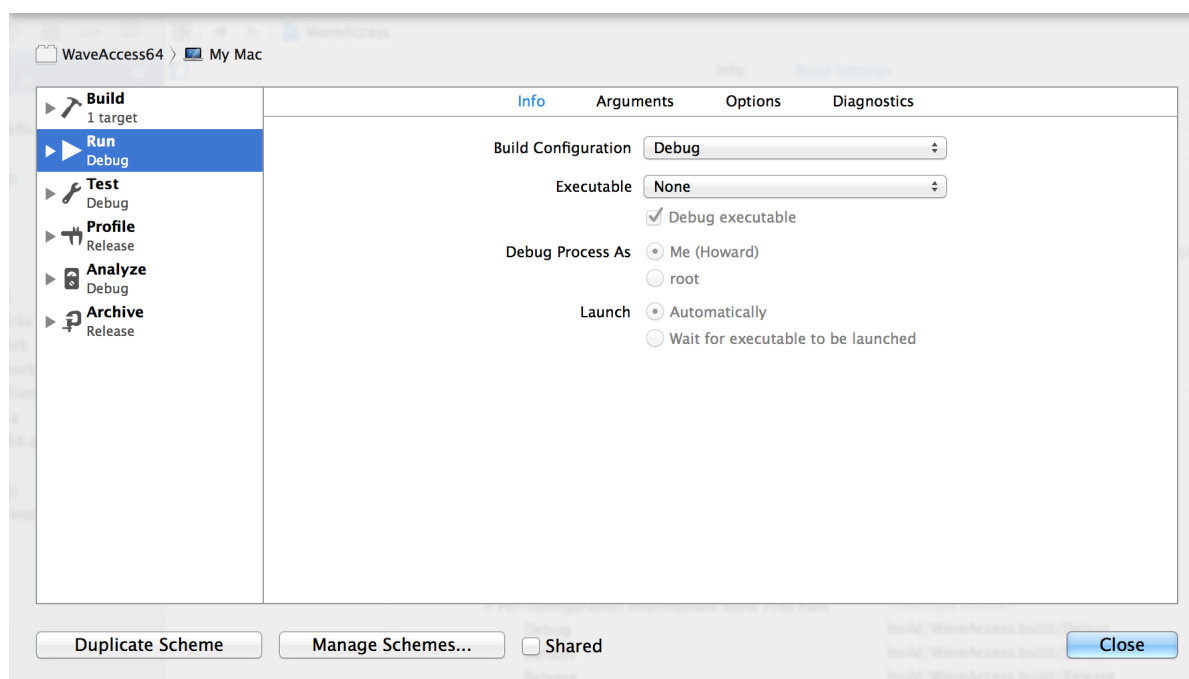


If you don't see the Active Scheme popup menu, choose View→Show Toolbar.

Because of an Xcode bug, the Active Scheme popup menu may not appear, even with the toolbar showing. As an alternative, you can use the Product→Scheme submenu.

The popup menu is really two popup menus in one. If you click “WaveAccess64”, you can select or edit the scheme. If you click “My Mac” you can select the “destination”. For Igor XOPs the destination is always “My Mac”.

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

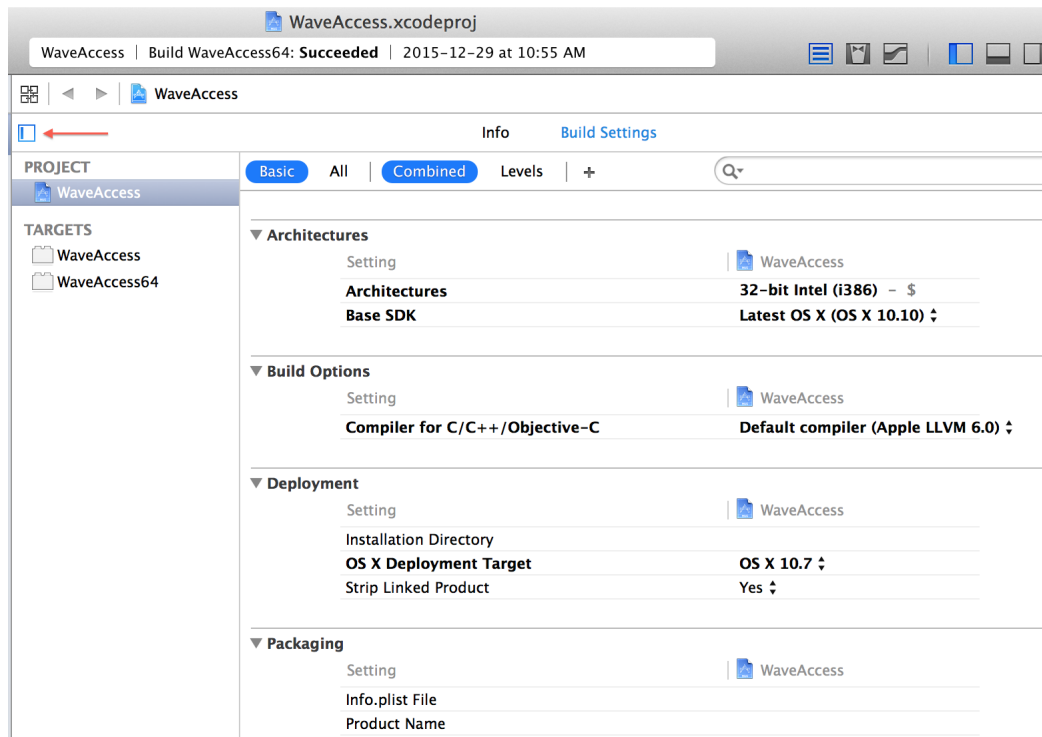


### Xcode Project Build Settings

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:



Xcode 10 supports 64-bit development only, not 32-bit development. If you are running Xcode 10 or later, you will need to adjust the following instructions accordingly.

*Architectures*: Set to “32-bit Intel”. We will override this for the 64-bit target.

*Base SDK*: Set to Latest MacOS.

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

*MacOS Deployment Target*: Set to 10.7 or later if 10.7 is not available.

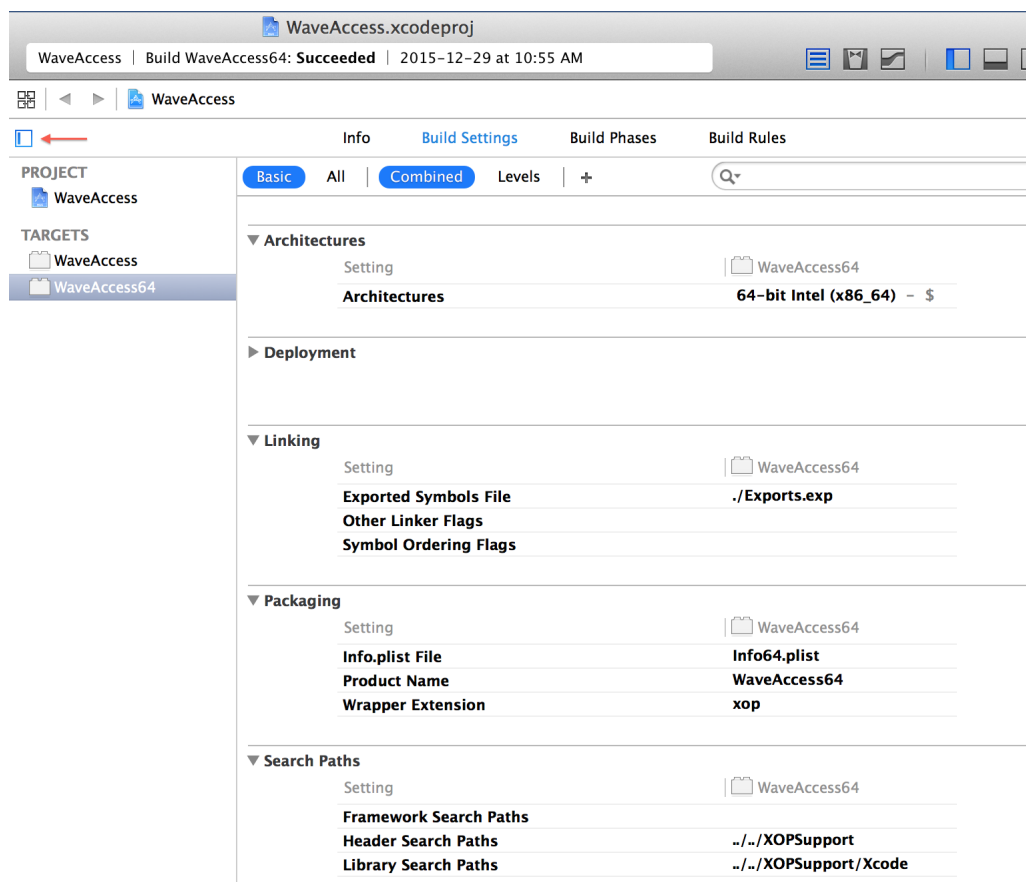
By setting the base SDK to “Latest MacOS”, we guarantee that the project will compile if you open it in a new Xcode version years from now. However this also lets you call Apple routines that are not available on the OS version that you happen to be running on, in which case your XOP will crash. For 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 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.

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 or WaveAccess64 under the TARGETS heading. Click the Build Settings tab and the Basic and Combined buttons:



Xcode 10 supports 64-bit development only, not 32-bit development. If you are running Xcode 10 or later, you will need to adjust the following instructions accordingly.

*Architectures:* Choose 32-bit Intel for the 32-bit target (WaveAccess) and 64-bit Intel for the 64-bit target (WaveAccess64).

*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.

*Info.plist File:* Set to Info.plist for the 32-bit target (WaveAccess) and Info64.plist for the 64-bit target (WaveAccess64).

*Product Name:* Set to WaveAccess for the 32-bit target (WaveAccess) and WaveAccess64 for the 64-bit target (WaveAccess64).

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

*Always Search User Paths:* Set to “No”.

*Header Search Paths:* Set to “../XOPSupport”. This allows #include statements to find the XOPSupport header files. The path is relative to the folder containing the .xcodeproj project file.

*Library Search Paths:* Set to “../XOPSupport/Xcode”. This allows the Xcode 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.

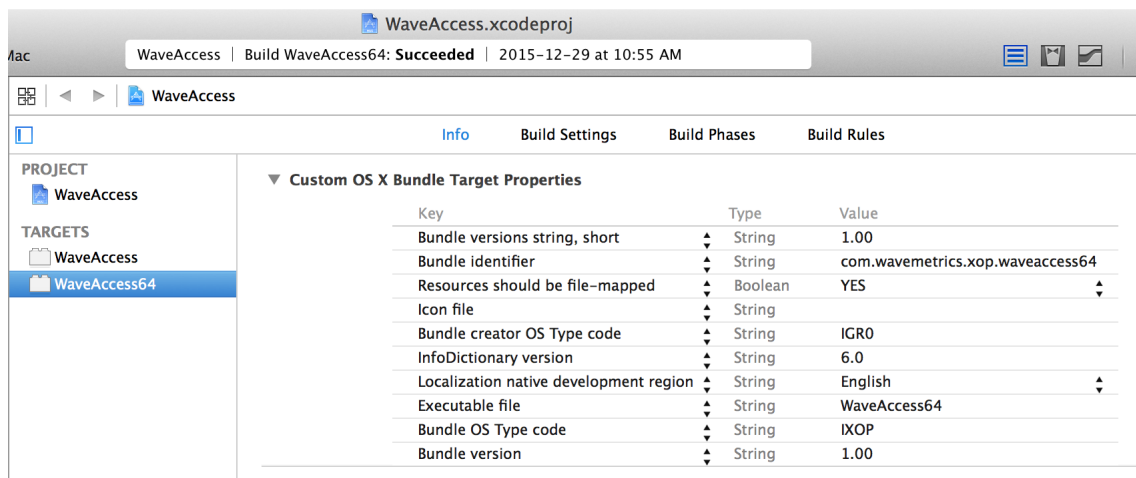
*Other Warning Flags:* Set to “-Wno-parentheses”. This turns off warnings for assigning the result of a function call to a variable in a conditional statement, such as:

```
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 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 or WaveAccess64 under the TARGETS heading. Click the Info tab:



Xcode 10 supports 64-bit development only, not 32-bit development. If you are running Xcode 10 or later, you will need to adjust the following instructions accordingly.

These settings are written to the project’s Info.plist or Info64.plist file which is copied to the XOP package folder 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 the WaveAccess XOP, the package name is WaveAccess.xop and the executable file, named WaveAccess, is located in the WaveAccess.xop package at Contents/MacOS/WaveAccess. For the WaveAccess64 XOP, the package name is WaveAccess64.xop and the executable file, named WaveAccess64, is located in the WaveAccess.xop package at Contents/MacOS/WaveAccess64.

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. This is explained further under **Check Your Info.plist File** on page 234.

*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 or com.wavemetrics.xop.waveaccess64. For your own XOPs you can make up an identifier of your choice.

## Chapter 3 — Development Systems

---

*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 MacOS. 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 “MacOS 10.9” as the base SDK for your project you will be able to use system routines defined for MacOS 10.9 but not routines defined for MacOS 10.10 because these require the 10.10 SDK. However, there is another problem. If you select, for example, “MacOS 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 MacOS, 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 7, we elected to set the Xcode projects up to use the “Latest MacOS” SDK. This prevents the problem described above where the project ceases to compile. But it introduces a new problem. It allows you to call MacOS 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.

### Missing SDK With Xcode

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 MacOS as explained under **Xcode Project Build Settings** on page 40.

### 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 Project Build Settings** on page 40.

You also may need to convert your XOP to use XOPMain as its main function. See **The XOPMain Function** on page 195 for details.

## Updating Xcode Projects for Xcode 10

This section explains updating XOP Toolkit 7 projects for Xcode 10 compatibility. If you are updating an XOP Toolkit 6 project, see **Xcode Project Changes For XOP Toolkit 7** on page 427.

Xcode 10 does not support 32-bit development. If you need to maintain the 32-bit version of your Macintosh XOP, you can not use Xcode 10. This section assumes that you do not want to maintain your 32-bit version.

Before starting, make a backup of your XOP projects.

If you have not already updated your project using XOP Toolkit 7.01 or later as explained under **XOP Toolkit 7 Upgrade Notes** on page 415 and **Xcode Project Changes For XOP Toolkit 7** on page 427, do that before moving to Xcode 10.

1. Open your project in Xcode 10.  
Xcode will display a dialog saying "32-bit MacOS targets are no longer supported".
2. Click OK.

### Showing the Projects and Targets List

1. In Xcode, choose View→Navigators→Show Project Navigator.
2. Select the top icon (e.g., "XOP1") in the Project Navigator. This shows build settings in the Xcode window central pane.

You should see a pane, called the Projects and Targets list, with sections PROJECT and TARGETS to the right of the Project Navigator pane. If you don't see the Projects and Targets list, click the square icon in the top/left corner of the project settings pane.

### Deleting the 32-bit Target

As noted above, it is assumed that you do not want to maintain the 32-bit version of your XOP. Xcode 10 does not support 32-bit development.

1. In Xcode, choose View→Navigators→Show Project Navigator.  
Select the top icon (e.g., "XOP1") in the Project Navigator.  
Select the 32-bit target (e.g., "XOP1") in the TARGETS list.  
Click the minus symbol in the bottom/left corner of the project settings pane to delete the 32-bit target.  
When prompted, click the Delete button.
2. Choose Product→Scheme→Manage Scheme.  
Select the 32-bit scheme (e.g., XOP1) in the list.  
Click the minus symbol in the bottom/left corner of the Manage Schemes window.  
When prompted, click the Delete button.  
Click Close.
3. In the Project Navigator pane, right-click the Info.plist icon and choose Delete. Do not delete Info64.plist.  
When prompted, click Move to Trash.
4. In the Project Navigator pane, right-click the libXOPSupport.a icon and choose Delete.  
If prompted, click Remove Reference.

### Updating to Recommended Project Settings

1. Choose View→Navigators→Show Issue Navigator.  
You should see an item in the Issue Navigator pane that says "Update to recommended settings". Click it. A window will open that lists settings to be updated.
2. Click Perform Changes.

### Changing the Deployment Target

1. Choose View→Navigators→Show Project Navigator.  
Select the top-level icon (e.g., XOP1).  
Select the project (e.g., XOP1) under PROJECT.
2. In the center pane, click "Info"  
Set MacOS Deployment Target to the earliest MacOS version that you want to support but it must be 10.9 or higher.
3. Click the 64-bit target (e.g., XOP1-64) under TARGETS and then click General.  
In the Deployment Info section, make sure that the Deployment Target is set the same as you set it for the project.

### Changing the “Always Search User Paths” Setting

1. Select the project (e.g., XOP1) under PROJECT again.  
Click Build Settings.
2. In the Search Paths section, set Always Search User Paths to No.
3. Click the 64-bit target (e.g., XOP1-64) under TARGETS. Make sure that the Always Search User Paths is set to No.

### Building the Project

Choose Project→Build. It should compile without errors though you may get warnings.

If you get an error that says “Could not decode input file using specified encoding”, in the Finder, overwrite the project’s InfoPlist.string file with the same-named file from an XOP Toolkit project. Then, in Xcode, replace the file’s contents with “// Localized strings go here”. Then choose Project→Build again.

If you get other errors you will need to debug them. There is no generic advise for errors at this point of the process.

### Handling Warnings

You may get warnings that you did not get before. This is because the "Updating to Recommended Project Settings" step turns on additional warnings. We recommend fixing warnings unless the fix is too complicated. Most warnings can be fixed with an appropriate cast. For example, change:

```
int len = strlen(str);  
  
to  
  
int len = (int)strlen(str);
```

You may see an intermittent "Enable Base Internationalization" warning. We are not aware of a reliable way to get rid of this warning.

### Debugging an XOP in Xcode

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:

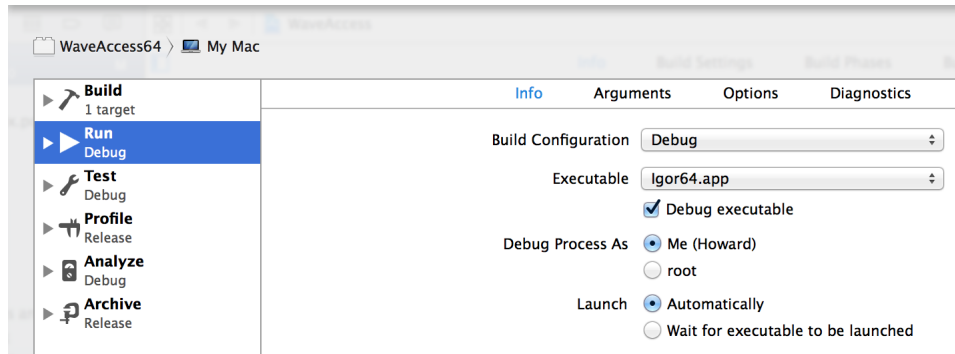
1. Choose Product→Scheme→Edit Scheme.
2. From the popup menu in the top/left corner of the window, choose WaveAccess or WaveAccess64.
3. Click Run in the list of actions on the left.
4. Click the Info tab.



5. Choose Debug from the Build Configuration popup menu.
6. Click the Executable popup menu, choose Other to display the Choose File dialog.
7. Select your Igor Pro application file and click the Choose button.

For Igor7 or later, this will be Igor.app for 32 bits and Igor64.app for 64 bits.

The scheme editor window should now look like this:



6. Click Close.
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 (e.g., /WaveAccess/Xcode/build/Debug/WaveAccess64.xop).

8. On the desktop, make an alias for that XOP and activate it as described under **Activating XOPs Using the Igor Extensions Folder** on page 6.
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.

## XOPs in Visual C++

On Windows, XOP Toolkit 7 supports development using Visual C++ 2015 and Visual C++ 2017. It will likely work with later versions of Visual C++, but see the caveats in the next section.

The community (free) edition of Visual C++ is fine for most 32-bit and 64-bit XOP development. Each sample project contains a VC folder. The VC folder, e.g., WaveAccess/VC, contains the Visual C++ solution file, e.g., WaveAccess.sln.

For instructions on converting an XOP Toolkit 6 project for use with XOP Toolkit 7, see **Converting Your Older Visual C++ Project** on page 434.

## Visual C++ Compatibility Issues

Microsoft's "Breaking Changes in Visual C++ 2015" document (<https://msdn.microsoft.com/en-us/library/bb531344.aspx>) warns against mixing code (applications and static or dynamic libraries that they call) compiled with different versions of Visual C++. Mixing can cause arcane compilation and linking

errors as well as runtime errors. These compatibility requirements cause serious headaches when moving to a new version of Visual C++.

However, Microsoft’s release notes for Visual C++ 2017 (<https://www.visualstudio.com/en-us/news/releasenotes/vs2017-relnotes>) say “No breaking changes between 2015 and 2017”. Based on this, and on our testing, it is safe to link Visual C++ 2017 code with Visual C++ 2015 libraries.

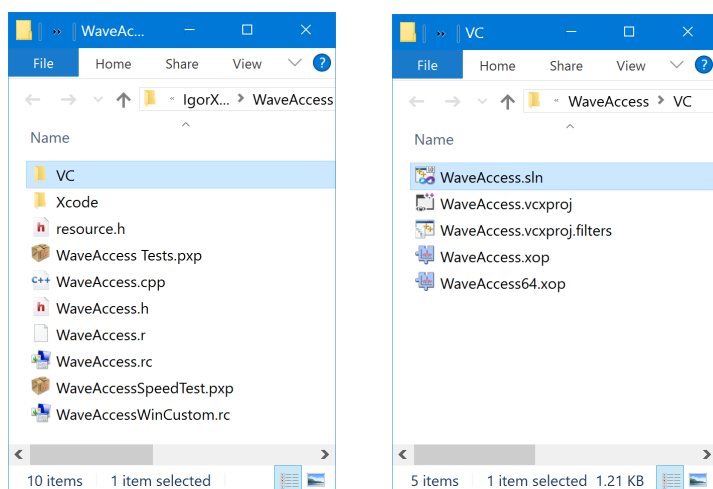
The XOP Toolkit 7 XOPSupport libraries are compiled using Visual C++ 2015. Consequently, XOPs should be compiled using Visual C++ 2015 or Visual C++ 2017.

If you must use a different version of Visual C++, you should recompile the XOPSupport libraries, as well as any other libraries that you use, using your version of Visual C++. To recompile XOPSupport, open XOPSupport.sln in Visual C++ and build the 32-bit and 64-bit release configurations to produce the XOPSupport.lib and XOPSupport64.lib libraries.

## XOP Projects in Visual C++

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 24.

We will use the WaveAccess sample XOP as a case in point. The WaveAccess folder is inside the IgorXOPs7 folder and looks like this:



The Visual C++ project files are inside the VC 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 XOPs are in the WaveAccess/VC folder.

WaveAccess.vcxproj is the Visual C++ project file and WaveAccess.sln is the Visual C++ “solution” file. To open the project, open the solution file in Visual Studio. 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.

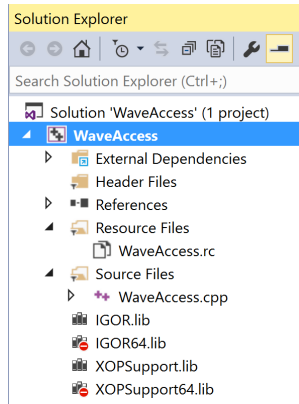
WaveAccess.rc is the main Windows resource file. It #includes resource.h and WaveAccessWinCustom.rc.

WaveAccess.rc contains standard Windows resources such as the version resource and menu and dialog resources, if any. Long ago, Visual C++ included a graphical resource editor which you used to create and edit standard Windows resources. The graphical resource editor created and managed the resource.h file. Then Microsoft removed the graphical resource editor from the free versions of Visual C++. Consequently, if you use a free version, you must edit these files using a text editor.

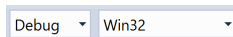
WaveAccessWinCustom.rc contains XOP-specific resources that you edit as text. Igor examines these custom resources to determine what operations, functions and menus the XOP adds, among other things. The use of these resource files is further explained under **XOP Resources** on page 55.

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

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 VC 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 VC folder. 64-bit XOPs are discussed in detail under **64-bit XOPs** on page 187.

## Visual C++ 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 VC 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 for both the 32-bit and 64-bit platforms 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 208 for details.

### C/C++ Properties / Command Line

*Additional Options:* Add `/utf-8`

This setting tells the compiler to compile literal strings as UTF-8. For background information, see **Text Encodings in Visual C++** on page 204.

### 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\VC). 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

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

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

## Debugging a Visual C++ 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, make a shortcut for the XOP and activate it as described under **Activating XOPs Using the Igor Extensions Folder** on page 6.
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.



## Igor/XOP Interactions

### Overview

The code on page 54 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
<b>XOPInit</b>	Stores the IORecHandle in a global variable which is used by all subsequent XOPSupport calls.
<b>SetXOPEntry</b>	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.
<b>RegisterOperation</b>	Tells Igor the syntax of the XOP1 external operation and the address of the function (ExecuteXOP1) that Igor must call to execute the operation.
<b>SetXOPResult</b>	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 are declared `extern "C"`. This is required in C++ code because all calls between Igor and the XOP use 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 a STR#,1101 resource.

### Creating Resources on Macintosh

You create Macintosh resources by editing a .r file as text and then letting Xcode compile the .r file. You should start with a .r file from a WaveMetrics sample XOP.

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 so that the resource compiler can compile XOP resources.

### 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 free 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"
```

The Visual C++ graphical resource editor 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.
    XOP_FEATURE_FLAGS, // Tells Igor about XOP features
    XOPI_RESERVED, // Reserved - must be 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.
XOP_FEATURE_FLAGS,     // Tells Igor about XOP features
XOPI_RESERVED,        // Reserved - must be 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 69 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, <b>Adding Operations</b> .
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, <b>Adding Functions</b> .
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 233.

A typical XOP will ignore most of these messages and respond to just a few of them.

As part of the response to a message from Igor, your XOP can pass a result code back to Igor using the **SetXOPResult** XOPSupport routine. The "Result" item in the following descriptions refers to this result code.

## Chapter 4 — Igor/XOP Interactions

---

A result code of 0 indicate success while a non-zero result code indicates an error. See **XOP Errors** on page 61 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.

<b>INIT</b>	<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>
<b>IDLE</b>	<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 <b>Handling Recursion</b> on page 67 for a discussion of recursion issues involving IDLE messages.</p>
<b>MENUITEM</b>	<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, <b>Adding Menus and Menu Items</b>, for details on how to respond to this message.</p>
<b>MENUENABLE</b>	<p>Tells your XOP to enable or disable its menu items.</p> <p>Arguments:    None.</p> <p>Result:        None.</p> <p>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 (<i>Macintosh</i>) or accelerator (<i>Windows</i>). This message gives your XOP a chance to enable, disable or change its menu items as appropriate.</p> <p>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.</p> <p>See Chapter 8, <b>Adding Menus and Menu Items</b>, for details on how to respond to this message.</p>
<b>CLEANUP</b>	<p>Tells your XOP that it is about to be closed and discarded from memory.</p>

Arguments: None.  
Result: None.

Your XOP should dispose any memory that it has allocated and do any other necessary cleanup.

Igor6 sends the CLEANUP message before Igor kills all experiment data. Igor7 and later send it after.

## 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 131 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 130 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

Tells your XOP that the user selected New Experiment from the File menu.

Arguments: None.  
Result: None.

	<p>When the user creates a new experiment your XOP should set its settings to their default state.</p> <p>Igor7 and later send the NEW message when Igor is about to quit. Igor6 does not send the NEW message at that time.</p>
<b>MODIFIED</b>	<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>
<b>CLEAR_MODIFIED</b>	<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>
<b>SAVE</b>	<p>This message is obsolete. Igor Pro will not send it.</p>
<b>LOAD</b>	<p>This message is obsolete. Igor Pro will not send it.</p>
<b>SAVESETTINGS</b>	<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 <b>Saving and Loading XOP Settings</b> on page 65 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 EXP_UNPACKED</p> <p>The save type will be one of the following which are defined in IgorXOP.h:</p> <p>SAVE_TYPE_SAVE SAVE_TYPE_SAVEAS SAVE_TYPE_SAVEACOPY SAVE_TYPE_STATIONERY</p>
<b>LOADSETTINGS</b>	<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.</p> <p>Argument 1:    Experiment type.</p> <p>Argument 2:    Load type.</p>

Result:           None.

If your XOP saved its settings in the experiment being opened you should reload those settings from the handle. If your experiment did not save its settings then the handle will be NULL and you should do nothing. See **Saving and Loading XOP Settings** on page 65 for details.

The handle that Igor passes to you comes from the experiment file and will be automatically disposed when Igor closes the experiment file so don't access it once you finish servicing the LOADSETTINGS message.

The experiment type will one of the following which are defined in IgorXOP.h:

EXP\_PACKED  
EXP\_UNPACKED

The load type will be one of the following which are defined in IgorXOP.h:

LOAD\_TYPE\_OPEN  
LOAD\_TYPE\_REVERT  
LOAD\_TYPE\_STATIONERY  
LOAD\_TYPE\_MERGE

## 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 0, this signifies success - that is, no error occurred. 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 64 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.
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 63.

### 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 provides 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 you return 0, this signifies success - that is, no error occurred. 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.

## Chapter 4 — Igor/XOP Interactions

---

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 return a non-zero error code 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 `GBLoadWaveX` 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. In Igor6, your data is stored in the Igor preferences. In Igor7 and later, your data is in a separate file in the XOPs folder of the Igor preferences folder. In both cases, the file is maintained by Igor and you should access it only via the `SaveXOPPrefsHandle` and `GetXOPPrefsHandle` functions.

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 often 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. Unless you take precautions, structures that you declare on one platform may not have the same layout as structures that you declare on the other platform, because of different structure alignment methods. There may also be differences in structure alignment between 32-bit and 64-bit code on the same platform.

To make sure that your structures have the same layout on all platforms and in 32 bits versus 64 bits, you need to tell the compiler how you want structures laid out. The sample XOPs on both Macintosh and Windows use two-byte alignment for all structures that are stored on disk or exchanged with Igor, thus satisfying this condition. This is done using the `pack pragma`. For details on this, see **Structure Alignment** on page 208.

The data that you store will often 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
<code>RESIDENT</code>	XOP wants to remain in memory indefinitely.
<code>TRANSIENT</code>	XOP wants to be closed and discarded.
<code>IDLES</code>	XOP wants to receive periodic <code>IDLE</code> messages.
<code>ADDS_TARGET_WINDOWS</code>	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 67 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. This scheme was devised in the previous millenium to circumvent the problem of different compilers using different conventions for passing parameter.

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.

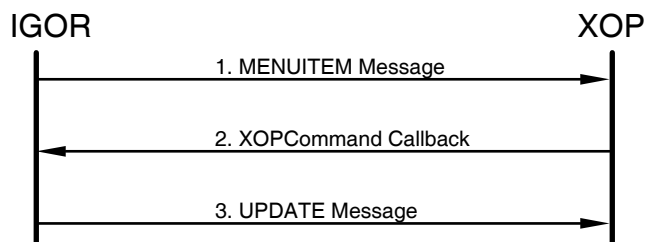
## Handling Recursion

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.

## Chapter 4 — Igor/XOP Interactions

---

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 **kXOPWindowMessageUpdate** message while it is still working on the **MENUITEM** message. 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.

Although this example involves the **MENUITEM** and **kXOPWindowMessageUpdate**, 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 **kXOPWindowMessageUpdate** message from Igor, 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 **kXOPWindowMessageUpdate** message before your initialization is complete. It is also possible to receive the **kXOPWindowMessageUpdate** 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.

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 share data. For example, your XOP may need to access and manipulate Igor waves, data folders, numeric variables, and string variables. Your external function or external operation may receive numeric and string parameters from Igor and return numeric and string results to it. When you make a callback to Igor, you may pass numeric and string parameters to Igor and receive numeric and string results from it.

When dealing with Igor objects, such as waves, data folders and variables, you must always use XOPSupport routines specific to the particular type of object. These routines are listed under **Routines for Accessing Waves** on page 253, **Routines for Accessing Data Folders** on page 300, and **Routines for Accessing Variables** on page 291.

Simple numeric parameters and results are represented using standard C data types such as int and double.

String parameters and results are represented using Handles. The Handle data type has its roots in the original Macintosh operating system and is still used in XOP programming for backward compatibility.

Handles and the routines used to manage them are described under **WM Memory XOPSupport Routines** on page 195.

## Igor/XOP Compatibility Issues

If you are targeting Igor Pro 8.00 or later, and you are creating a new XOP or updating an XOP already updated using XOP Toolkit 7.01 or later, you should use XOP Toolkit 8. See **Choosing Which XOP Toolkit to Use** on page 5 for details.

With XOP Toolkit 7, in most cases you can easily support Igor6 (6.20 or later) and Igor7 or later with one XOP. An exception is if you make frequent use of non-ASCII text. In that case, you may be better off maintaining separate Igor6 and Igor7 versions of your XOP - see **Text Encodings** on page 201 for details.

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 7 require Igor Pro 6.20 or later. Consequently the XOPMain function in each XOP must contain code like this:

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

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

For example, if your XOP uses features added to Igor in version 7.00, you must restrict your XOP to running with Igor Pro 7.00 or later. Put the following in your XOP's XOPMain function:

```
if (igorVersion < 700) {           // Requires Igor Pro 7.00 or later.
    SetXOPResult(IGOR_OBSOLETE);
    return EXIT_FAILURE;
}
```

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 7 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 parameter.
/F=(f1, f2, . . .)	A flag with a set of numeric parameters.
/F=[f1, f2, . . .]	A flag with a set of numeric indices.
/F={f1, f2, . . .}	A flag with a list of parameters of possibly different types.

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. If possible, choose a syntax similar to an existing Igor 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” or “Igor Extensions (64-bit)” 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.

When the user invokes one of these operations from Igor’s command line or from a procedure, if the XOP was not already loaded, Igor loads the XOP into memory and calls its XOPMain function. 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.

If the operation is thread-safe, the XOPC entry for it must include the threadSafeOp flag.

### 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. If your operation is thread-safe, set the threadSafeOp bit in the XOPC resource for the operation.
7. 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 should create your own template experiment by duplicating that file or a template file from another sample XOP.

“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 82.

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 77. 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 which describes the operation's syntax. 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 70.

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 70.

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 vs 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 70.

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 provided.

## 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 74. 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>"
```

If your operation is thread-safe, include the /TS flag in the ParseOperationTemplate command.

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.
```

## Chapter 5 — Adding Operations

---

```
int key1Encountered;
double key1Num;
Handle key1StrH;
int key1ParamsSet[2];

// Parameters for key2 keyword group.
int key2Encountered;
char key2Name[MAX_OBJ_NAME+1];
waveHndl key2WaveH;
int key2ParamsSet[2];

// These are postamble fields that Igor sets.
int calledFromFunction;    // 1 if called from a user-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.

## Chapter 5 — Adding Operations

---

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.

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 70.

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 **WMHandToHand** 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 **WMGetHandleSize** 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 233 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 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.

### 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.

## Chapter 5 — Adding Operations

---

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.

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 70.

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:

## Chapter 5 — Adding Operations

---

- 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.

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 249.

## 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 208 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

## Chapter 5 — Adding Operations

---

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.

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 92.

### 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 131.

### 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 143.

### 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 209.

### 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 struc-

ture 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 74, 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 required in C++ code because all calls between Igor and the XOP use C calling conventions. See **C++ Code Called From C Code** on page 217 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 94.

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.

## Chapter 5 — Adding Operations

---

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:

```
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 175.

## 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 79. See **Creating Starter Code** on page 74 and **Updating Starter Code** on page 95.
- 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 **WMHandToHand** or **GetCStringFromHandle**.
- String handles are not C strings. Use **WMGetHandleSize** 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 233 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 many cases they run 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.

External functions can be thread-safe.

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.

The SimpleFit sample XOP illustrates implements a curve-fitting function and the **Guided Tour** on page 19 shows how to modify it to fit a different function.

## 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, NT\_I32 and NT\_I64, 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 70.

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 70.

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 104.

## Chapter 6 — Adding Functions

---

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 114.

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.

If your XOP is thread-safe, you must set the F\_THREADSAFE bit.

### 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 208 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 required in C++ code because all calls between Igor and the XOP use 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.

## Chapter 6 — Adding Functions

### 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.

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 114. 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 61 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 70.

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 70.

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 107.

## 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 70.

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 116.

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;

    p->result = NULL;           // Must be set to a valid handle or NULL

    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 = WMGetHandleSize(p->str2); // length of string 2
    len3 = WMGetHandleSize(p->str3); // length of string 3
    str1 = WMNewHandle(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:
    WMDisposeHandle(p->str2);     // Get rid of input parameters. (OK if NULL)
    WMDisposeHandle(p->str3);     // Get rid of input parameters. (OK if NULL)
    p->result = str1;             // Must be a valid handle or NULL

    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 **WMGetHandleSize** 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 233 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:** Before your function returns, you must store a valid handle in the result field or set it to `NULL`.

**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
```

## Chapter 6 — Adding Functions

---

```
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.

## Chapter 6 — Adding Functions

---

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 209.

### 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 131.

### 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 143.

## 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
        {                        // Parameter types
            NT_FP64 | FV_REF_TYPE,
            NT_FP64 | NT_CMPLX | FV_REF_TYPE,
            HSTRING_TYPE | FV_REF_TYPE,
```

```
    },
  }
};
```

Second, in the parameter structure passed to the XFUNC, the fields corresponding to pass-by-reference parameters must be pointers to doubles or pointers to Handles. For example:

```
#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 = WMNewHandle(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.

Support for thread-safe external functions was greatly improved in Igor Pro 6.20. Since XOPs created with XOP Toolkit 7 require Igor Pro 6.20 or later, this section assumes that you are running with 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.

If you mark an external function as thread-safe, Igor allows that function to be called from user-defined functions marked as thread-safe. The user-defined function and your external function may then be called from Igor's main thread or from an Igor pre-emptive thread.

You can not call an XOPSupport routine when running in an Igor pre-emptive thread unless the routine is marked as thread-safe. The documentation for each XOPSupport routine states whether it is thread-safe or not.

You can call a thread-safe XOPSupport routine only from Igor's main thread or from an Igor pre-emptive thread, not from a private thread that you create on your own.

### 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.

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 116.
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 175.

## 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 **WM Memory XOPSupport Routines** on page 195 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, use the XOPSupport routines, provided in XOPWaveAccess.c and listed in Chapter 15, to get and set wave data and properties.

*You also must not treat a wave handle as a WM memory handle by passing it to normal handle routines such as WMGetHandleSize or as a Macintosh native handle by passing it to a Macintosh native routine such as GetHandleSize.*

Instead, use the XOPSupport routines, provided in XOPWaveAccess.c and listed in Chapter 15. See **Change to Wave Handles** on page 417 for details.

## Routines for Accessing Waves

The XOPSupport routines for dealing with waves are described in detail under **Routines for Accessing Waves** on page 253. Here is a summary of the commonly-used routines.

### Getting a Handle to a Wave

Routine	Description
<b>MakeWave</b>	Makes 1D waves.
<b>MDMakeWave</b>	Makes waves of dimension 1 through 4.
<b>FetchWave</b>	Returns a handle to a wave in the current data folder given its name.
<b>GetDataFolderObject</b>	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.
WaveNoteCopy	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
<b>WaveData</b>	Returns pointer to the start of wave's data.
<b>MDAccessNumericWaveData</b>	Returns the offset to wave data.
<b>MDSetNumericWavePointValue</b>	Sets the value of a single element of wave.
<b>MDStoreDPDataInNumericWave</b>	Sets all wave data from a double-precision buffer.
<b>MDSetTextWavePointValue</b>	Sets the value of a single element of text wave.
<b>SetTextWaveData</b>	Sets the entire contents of a text wave.

**Wave Reference Counting**

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

**Obsolete Routines**

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

**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 10 floating point and integer numeric data 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_I64	128	8	Signed 64-bit integer (SInt64). Requires Igor7 or later.
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_I64   NT_UNSIGNED	128+64	8	Unsigned 64-bit integer (UInt64). Requires Igor7 or later.
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.

The 64-bit integer types require Igor Pro 7.00 or later.

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).

When storing into an integer wave, **MDSetNumericWavePointValue** truncates the value that you are storing. If you want, you can do rounding before calling **MDSetNumericWavePointValue**.

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.

**MDGetNumericWavePointValue** and **MDSetNumericWavePointValue** use doubles which can not precisely represent integers of magnitude greater than  $2^{53}$ . If you must use 64-bit integer waves and require full precision for very large values, you must use **The Direct Access Method** (see page 126) or these routines which require Igor Pro 7.03 or later:

<b>MDGetNumericWavePointValueSInt64</b>	<b>MDSetNumericWavePointValueSInt64</b>
<b>MDGetNumericWavePointValueUInt64</b>	<b>MDSetNumericWavePointValueUInt64</b>

See **64-bit Integer Issues** on page 215 for further discussion.

### 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*)WMNewPtr(numBytes);
if (dPtr==NULL)
    return NOMEM;
if (result = MDGetDPDataFromNumericWave(waveH,dPtr)) { // Get copy.
    WMDisposePtr((Ptr)dPtr);
    return result;
}
dp = dPtr;
<Use dp to access wave data.>
result = MDStoreDPDataInNumericWave(waveH,dPtr);        // Store data.
WMDisposePtr((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 127.

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.

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 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.

The temporary storage access method uses doubles which can not precisely represent integers of magnitude greater than  $2^{53}$ . If you must use 64-bit integer waves and require full precision for very large values, you must use **The Direct Access Method**, described in the next section. See **64-bit Integer Issues** on page 215 for further discussion.

### 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 127. See Chapter 15 for an example illustrating the use of MDAccessNumericData.

There are three 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 10 numeric types listed above. You need a different type of pointer to deal with each of these data types. This is illustrated in the example for `MDAccessNumericWaveData` in the WaveAccess sample XOP. If you are using C++, you can use a template to handle all of the data types.

The 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.

The last difficulty involves resizing the wave which you do by calling **ChangeWave**, **MDChangeWave** or **MDChangeWave2**. When you increase the size of a wave, the memory manager may be required to move the block of memory holding the wave’s data in order to enlarge it. If that happens, any pointer to the wave data that you previously acquired, such as the `dp` variable shown above, becomes a dangling pointer - pointing to something that is no longer there. You must set the pointer again to make sure that it points to the waves data. See **Dangling Pointers** on page 228 for further discussion.

### 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
MacOS 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.

## Chapter 7 — Accessing Igor Data

---

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
<code>dimensionSizes[ROWS]</code>	Number of rows in a column.
<code>dimensionSizes[COLUMNS]</code>	Number of columns in a layer. Zero for 1D waves.
<code>dimensionSizes[LAYERS]</code>	Number of layers in a chunk. Zero for 1D and 2D waves.
<code>dimensionSizes[CHUNKS]</code>	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, which treat the element as a C string, will treat the 0 byte as the end of the string.

The organization of wave text data in memory is an implementation detail of Igor and your code can not depend on it.

Igor6 stored 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 contained another handle in which Igor stores indices. The indices told Igor where the text for any given element in the text wave began.

As of Igor7, Igor stores each text wave element in a separate block of memory referenced by a separate pointer. This makes insertions and deletions much faster at the expense of greater memory usage.

You should not attempt to access the text data or the text indices directly. Instead, use the XOPSupport routines provided 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 WMGetHandleSize. 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 233.

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  
 X Units

Data  
 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	Column Index
		0.0 m	2.5 m	5.0 m	Y Index and Units
0	0.0 s	3.57 v	8.24 v	0.14 v	
1	0.1 s	3.21 v	8.44 v	0.26 v	
2	0.2 s	2.97 v	8.76 v	0.33 v	
3	0.3 s	2.73 v	8.94 v	0.47 v	
4	0.4 s	2.44 v	9.31 v	0.52 v	
5	0.5 s	2.03 v	9.45 v	0.55 v	

Row Index      X Index      Data      Data      Data  
                     X Units      Data Units      Data Units      Data Units

Data full scale = -10, 10

You can set both the X (row) and Y (column) units and scaling using a **SetScale** command from within Igor and using the **MDSetWaveUnits** and **MDSetWaveScaling** XOPSupport routines from an XOP. If we extend to three dimensions, we add Z (layer) units and scaling and still use the same routines to set them.

Note that, as you go from 1D to 2D, you still have only one data units string and one data full scale. This is true no matter how many dimensions 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 while your XOP code is running only 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.

You can prevent the killing of a wave by responding to the **OBJINUSE** message, as described in the next section.

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

### 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 131, 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 59 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 130, 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.

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 reference 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 131 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**.

## Chapter 7 — Accessing Igor Data

---

```
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.
```

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 94. 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 **WMGetHandleSize** 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 233.

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 291. Here is a summary of the commonly-used routines.

### Creating Variables

Routine	Description
<b>Variable</b>	Creates an Igor numeric or string variable. This routine is antiquated. The following routines are recommended.
<b>SetIgorIntVar</b>	Creates if necessary and then sets a numeric variable to an integer value.
<b>SetIgorFloatingVar</b>	Creates if necessary and then sets a numeric variable to a floating point value.
<b>SetIgorComplexVar</b>	Creates if necessary and then sets a numeric variable to a complex floating point value.
<b>SetIgorStringVar</b>	Creates if necessary and then sets a string variable.

### Getting Variable Contents

Routine	Description
<b>FetchNumVar</b>	Gets the value of a numeric variable.
<b>FetchStrVar</b>	Gets up to 255 characters from a string variable.
<b>FetchStrHandle</b>	Gets any number of characters from a string variable.
<b>GetDataFolderObject</b>	Gets the value of numeric or string variable.

### Setting Variable Contents

Routine	Description
<b>StoreNumVar</b>	Sets the value of a numeric variable.
<b>SetIgorIntVar</b>	Creates if necessary and then sets a numeric variable to an integer value.
<b>SetIgorFloatingVar</b>	Creates if necessary and then sets a numeric variable to a floating point value.
<b>SetIgorComplexVar</b>	Creates if necessary and then sets a numeric variable to a complex floating point value.
<b>SetIgorStringVar</b>	Creates if necessary and then sets a string variable.
<b>SetFileLoaderOutputVariables</b>	Specialized for file-loader XOPs.
<b>SetFileLoaderOperationOutputVariables</b>	Specialized for file-loader XOPs. Called from direct external operations only.
<b>SetDataFolderObject</b>	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 94.

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
<b>GetRootDataFolder</b>	Returns a handle to the root data folder.
<b>GetCurrentDataFolder</b>	Returns a handle to the current data folder.
<b>GetNamedDataFolder</b>	Returns a handle to a data folder, given its name or path.
<b>GetWavesDataFolder</b>	Returns a handle to the data folder containing a particular wave.
<b>GetParentDataFolder</b>	Returns a handle to the parent of the specified data folder.
<b>GetNumChildDataFolders</b>	Returns the number of child data folders within a particular data folder.
<b>GetIndexedChildDataFolder</b>	Returns a handle to the specified data folder within a particular data folder.
<b>GetDataFolderByIDNumber</b>	Returns a handle to a data folder identified by an ID number that you previously obtained from <b>GetDataFolderIDNumber</b> .

#### Creating and Killing a Data Folder

Routine	Description
<b>NewDataFolder</b>	Creates a new data folder.
<b>DuplicateDataFolder</b>	Duplicates an existing data folder and its contents.
<b>KillDataFolder</b>	Kills a data folder.

#### Accessing Objects in a Data Folder

Routine	Description
<b>GetNumDataFolderObjects</b>	Returns the number of objects (waves, numeric variables, string variables, data folders) in a particular data folder.
<b>GetIndexedDataFolderObject</b>	Returns the name of a particular object.
<b>GetDataFolderObject</b>	Returns information about a particular object.
<b>SetDataFolderObject</b>	Sets information about a particular object.
<b>KillDataFolderObject</b>	Kills a particular object.
<b>DuplicateDataFolderObject</b>	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

## Chapter 7 — Accessing Igor Data

---

**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 131 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 133 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 148. 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 Menus in XOP Toolkit 7** on page 421.

### Summary of XOPSupport Menu Routines

Here are the XOPSupport menu routines:

Routine	What It Does
<b>ResourceToActualMenuID</b>	Given the ID of a MENU resource in the XOP's resource fork, returns the actual menu ID of that menu in memory.
<b>ActualToResourceMenuID</b>	Given the ID of a menu in memory, returns the resource ID of the MENU resource in the XOP's resource fork.
<b>ResourceToActualItem</b>	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.
<b>ActualToResourceItem</b>	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.
<b>SetIgorMenuItem</b>	Enables or disables a built-in Igor menu item.
<b>XOPActualMenuIDToMenuRef</b>	Given the actual menu ID of an XOP menu in memory, returns a menu reference for the XOP menu.
<b>XOPResourceMenuIDToMenuRef</b>	Given the ID of a MENU resource in the XOP's resource fork, returns a menu reference for that menu.
<b>XOPGetMenuInfo</b>	Returns information about a menu.
<b>XOPCountMenuItems</b>	Returns the number of items in a menu.
<b>XOPShowMainMenu</b>	Makes a menu appear in the menu bar if it was previously hidden.
<b>XOPHideMainMenu</b>	Removes the menu from the menu bar if it was previously showing.
<b>XOPGetMenuItemInfo</b>	Returns information about a menu item.
<b>XOPGetMenuItemText</b>	Returns the text from an XOP menu item.
<b>XOPSetMenuItemText</b>	Sets the text of an XOP menu item.
<b>XOPAppendMenuItem</b>	Adds a menu item to the end of an XOP menu.
<b>XOPInsertMenuItem</b>	Inserts a menu item in an XOP menu.
<b>XOPDeleteMenuItem</b>	Removes a menu item from an XOP menu.
<b>XOPDeleteMenuItemRange</b>	Removes a range of menu items from from an XOP menu.
<b>XOPEnableMenuItem</b>	Enables a menu item.
<b>XOPDisableMenuItem</b>	Disables a menu item.
<b>XOPCheckMenuItem</b>	Checks or unchecks a menu item.
<b>XOPFillMenu</b>	Inserts multiple items into an XOP menu.

Routine	What It Does
<b>XOPFillMenuNoMeta</b>	Inserts multiple items into an XOP menu.
<b>XOPFillWaveMenu</b>	Inserts multiple wave names into an XOP menu.
<b>XOPFillPathMenu</b>	Inserts multiple symbolic path names into an XOP menu.
<b>XOPFillWinMenu</b>	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 MenuHandle. When running with Igor Pro 6 on Windows an XOPMenuRef is a handle created internally by Igor that emulates a Macintosh MenuHandle. When running with Igor Pro 7 an XOPMenuRef is 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 151.

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,                        // menuID: Add item to menu with ID=8.
        "MenuXOP1 Misc1",        // itemText: This is text for added menu item.
        0,                        // subMenuResID: This item has no submenu.
        0,                        // itemFlags: Flags field.
    }
};

1100 XMI1                          // Windows, in MenuXOP1WinCustom.rc.
BEGIN
    8,                             // menuID: Add item to menu with ID=8.
    "MenuXOP1 Misc1\0",           // itemText: This is text for added menu item.
    0,                             // subMenuResID: This item has no submenu.
    0,                             // itemFlags: Flags field.
    0,                             // 0 required to terminate the resource.
END
```

The first field, called `menuID`, is the menu ID of the built-in Igor menu to which the menu item should be attached. The file `IgorXOP.h` gives the menu ID's of all Igor menus.

The second field, called `itemText`, is a string containing the text for the added menu item.

The third field, called `subMenuResID`, is the resource ID of the MENU resource in the XOP's resource fork for the submenu to be attached to the menu item or 0 for no submenu.

The fourth field, called `itemFlags`, consists of bitwise flags defined in `XOP.h` which tell Igor under what conditions the menu item should be enabled or disabled. See **Enabling and Disabling XOP Menu Items in Igor Menus** on page 152.

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.

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.

## Chapter 8 — Adding Menus and Menu Items

---

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 **ResourceToActualMenuID** 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 kXOPWindowMessageCut, kXOPWindowMessageCopy, and kXOPWindowMessagePaste 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 exists in the current data folder and automatically disable it if no waves exist in the current data folder.

For another example, if the ITEM\_REQUIRES\_GRAPH bit is set, Igor will enable your menu item only if the target window is a graph.

## Enabling and Disabling Igor Menu Items

If your XOP adds a window to Igor, you might want to enable and disable Igor menu items such as Cut and Copy when your window is active. This section describes how to do this.

The **SetIgorMenuItem** XOPSupport routine allows the XOP to enable or disable a built-in Igor menu item without referring to specific menu IDs or item numbers. Here is a description:

```
int
SetIgorMenuItem(message, enable, text, param)
int message;           // an Igor message code
int enable;            // 1 to enable the menu item, 0 to disable it
char* text;            // pointer to a C string or NULL
int param;             // normally not used and should be 0
```

For example, if the user selects the Copy menu item, Igor sends the XOP the `kXOPWindowMessageCopy` message. If the XOP wants to enable the Copy menu item, it would call Igor as follows in response to the `MENUENABLE` message:

```
if (IsIgorWindowActive((xopWindowRef)) // Is XOP window active?
    SetIgorMenuItem(kXOPWindowMessageCopy, 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 (IsIgorWindowActive((xopWindowRef)) // Is XOP window active?
    SetIgorMenuItem(kXOPWindowMessageUndo, 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 (IsIgorWindowActive((xopWindowRef)) { // Is XOP window active?
        SetIgorMenuItem(kXOPWindowMessageCopy, 1, NULL, 0);
        SetIgorMenuItem(kXOPWindowMessageCut, 1, NULL, 0);
    }
    .
    .
}
```



## Adding Windows

### Overview

This chapter applies to Igor7 or later, not to Igor6.

The Igor7 XOP window protocol is completely different from the Igor6 protocol. Igor6 XOPs that create windows do not work in Igor7 and vice versa. If you are porting an existing XOP to Igor7, see **Changes to XOP Window Programming in Igor Pro 7** on page 172 for an overview of the changes.

An XOP can add one or more windows to Igor. It can also add subwindows to built-in Igor window types.

Adding an XOP window to Igor is much more involved than adding an operation or a function. It involves both platform-independent programming to communicate with Igor and platform-dependent programming to implement the user interface. If you are writing a Macintosh XOP, you will need to be comfortable with Objective-C++ and the Macintosh Cocoa API. If you are writing a Windows XOP, you will need to be comfortable with the Windows API (formerly called the Win32 API).

Programming with windows raises 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. If it is possible to achieve your objective using a built-in Igor window, this will be much easier than creating an XOP window.

The integration of XOP windows into Igor requires coordination between the XOP, Igor, the Qt framework on which Igor7 is based, and the operating system. These components do not always work harmoniously. Consequently perfect behavior is not always possible, but it is possible to create a useable system. The main known issues are listed under **XOP Window Issues** on page 171.

Over time, changes to Igor, the Qt framework or the operating system may require changes to your XOP. You should not undertake XOP window programming unless you are prepared to deal with such changes.

In this chapter we use the term "window XOP" as shorthand for an XOP that adds one or more windows to Igor. We use "XOP window protocol" to mean the set of rules by which an XOP adds a window to Igor.

### The WindowXOP1 Sample XOP

The WindowXOP1 sample XOP illustrates XOP window programming.

**NOTE:** The WindowXOP1 sample XOP is not shipped with XOP Toolkit 7. If you want the source code please contact WaveMetrics support.

To write your own window XOP, use the "Create New XOP Project.pxp" experiment to make a clone of the WindowXOP1 project folder. Then modify the resulting project to implement the desired behavior.

### Types of XOP Windows

An XOP can add a one-of-a-kind window, like Igor's Data Browser, or a type of target window, like Igor's graph window type. The WindowXOP1 sample XOP illustrates both.

An XOP target window can be embedded in a built-in Igor window, like adding an Igor graph subwindow to an Igor panel. In this case the XOP window is a subwindow.

### XOP Window Documentation

Very few XOP programmers have the need or capability to create a window XOP. Consequently, to reduce clutter for the vast majority of XOP programmers, documentation for window-oriented XOPSupport routines is omitted from Chapter 15, **XOPSupport Routines**. The documentation for these routines can be found in the following XOPSupport files:

XOPWindows.c

XOPContainers.c

Also the window-oriented messages that Igor sends to a window XOP are not listed in Chapter 4, **Igor/XOP Interactions**. Instead they are listed in these XOPSupport files:

XOPWindows.h

XOPContainers.h

## Guided Tour of WindowXOP1

The purpose of this section is to give you feel for XOP window programming and get you oriented in preparation for writing your own XOP. Execute the steps below.

### Setting the Project Up

#### 1. Obtain the WindowXOP1 project folder from WaveMetrics support.

Send an email to support@wavemetrics.com requesting the WindowXOP1 sample XOP project. Please include in your request a description of your planned XOP. Mention what operating systems you plan to support.

#### 2. Unzip the WindowXOP1.zip file into your IgorXOPs7 folder.

#### 3. Open the WindowXOP1 folder in the Macintosh Finder or in Windows Explorer.

There are four kinds of source files.

The files in the WindowXOP1 folder whose names start with XOP are generic files that should work for any XOP. You probably will not need to modify them.

The files in the WindowXOP1 folder whose names start with WindowXOP1 are application-specific files that you will need to modify for your own application.

The files in the Mac folder are application-specific Objective-C++ files used on Macintosh only.

The files in the Windows folder are application-specific files used on Windows only.

#### 4. Open the Macintosh or Windows project file in your development system.

#### 5. If you are on Macintosh, set the project settings so that the compiled XOP will be written inside the Xcode folder in your WindowXOP1 folder.

You can skip this step if you are an experienced Xcode programmer and you are comfortable with where Xcode stores compiled output by default.

This is necessary because, ironically, Xcode does not store "project settings" in the project file so they must be set by each XOP programmer.

Access the project settings by choosing File→Project settings. Click the Advanced button. In the resulting dialog, set the Build Location to "Legacy".

Click Done to leave the Advanced dialog and then click Done again to leave the Project Settings dialog.

#### 6. Compile the 64-bit version of the XOP.

#### 7. Activate the 64-bit version of the XOP.

Put an alias or shortcut for the WindowXOP1-64.xop file in the "Igor Extensions (64-bit)" folder of your

"Igor Pro 7 User Files" folder.

8. **Configure your Xcode or Visual C++ project to set your installed 64-bit version of Igor7 as the host application for debugging.**

See **Development Systems** on page 35 for details. Search for "Debugging" until you find the instructions relevant to your development system.

9. **Start debugging the WindowXOP1 XOP.**

See **Development Systems** on page 35 for details. Search for "Debugging" until you find the instructions relevant to your development system.

WindowXOP1 creates a window when Igor is launched. This is done in the XOPMain function. The window created is a one-of-a-kind window (type XOPWIN), not a target window (type XOPTARGWIN).

The bottom of the WindowXOP1 window shows debugging information used by WaveMetrics.

## Using the WindowXOP1 Window

1. **Activate the window and click the Print button to the right of the Time readout.**

WindowXOP1 prints the time in the history area.

2. **Enter "Hello" in the edit control and click the associated Print button.**

WindowXOP1 prints "Hello" in the history area.

3. **Drag and resize the window.**

It works like a normal window.

4. **Click the close box to hide the window.**

The XOP programmer can tell Igor whether the close box should hide or kill the window. In WindowXOP1, the close box hides the one-of-a-kind window created at startup but kills target windows (demonstrated below).

5. **Choose Windows→Other Windows→WindowXOP1.**

The window is made visible.

6. **Choose Windows→Hide to hide the window again.**

## Creating a WindowXOP1 Target Window

The window that we played with in the preceding section is a one-of-a-kind window (type XOPWIN), analogous the Igor's Data Browser. Now we will create a target window (type XOPTARGWIN), analogous to an Igor graph or table.

1. **Execute this command in Igor's command line:**

```
NewWindowXOP1Window
```

A new window is created. Unlike the original one-of-a-kind window created at launch time, this is a target window, like a graph or table. You can create as many WindowXOP1 target windows as you want.

Notice that a WindowXOP1 menu appeared in the menu bar. This is WindowXOP1's "slot" menu and is analogous to the Graph or Table menu that appears when an Igor graph or table window is active.

2. **Click the Close box.**

Igor displays the Close Window dialog.

3. **Click the Save button.**

The window is saved in the built-in procedure window as a window recreation macro and is killed.

4. **Open the Procedure window, scroll up and read the window recreation macro.**

5. **Close the Procedure window.**

**6. Choose Windows→Other Macros→WX0.**

The window is recreated.

The next several steps are just to show that the WindowXOP1 window behaves mostly like any other Igor window.

**7. Click in the edit control and enter "Hello" without the quotes.**

**8. Choose Edit→Select All.**

**9. Choose Edit→Cut.**

**10. Choose Edit→Undo.**

**11. Move the WX0 window so that it is partially offscreen.**

**12. Choose Windows→Control→Retrieve Window.**

**13. Press the Shift key and choose Windows→Hide.**

The WX0 window is hidden.

### Creating a WindowXOP1 Subwindow

Just as you can embed, for example, a graph as a subwindow in a control panel, you can embed an XOP window in a panel as subwindow. That's what we will do next.

**1. Execute this command in Igor's command line:**

```
NewPanel/W=(200,50,750,450)
```

**2. Execute:**

```
NewWindowXOP1Window/HOST=#
```

A new subwindow is created in the panel window. Its behavior is very similar to the behavior of a graph subwindow.

**3. Choose Panel→Show Tools.**

**4. Click the Drawing icon (second from top) in the tool palette.**

**5. Click the WindowXOP1 subwindow.**

The subwindow is selected.

**6. Click the selection frame on the periphery of the WindowXOP1 subwindow.**

We are in subwindow layout mode.

**7. Click one of the edges of the layout frame and drag the XOP subwindow so that it is centered in the panel window.**

Laying out an XOP subwindow works the same as laying out an Igor subwindow.

**8. Press Cmd-T (Macintosh) or Ctrl-T (Windows) to hide the tool palette.**

It does not work. This is one of those things where XOP subwindows don't work exactly like Igor subwindows.

**9. Click in the body of the panel window to leave subwindow layout mode.**

**10. Click in the body of the panel window again to deselect the XOP subwindow.**

**11. Press Cmd-T (Macintosh) or Ctrl-T (Windows) to hide the tool palette.**

This time it works.

**12. Click the panel Close box.**

Igor displays the Close Window dialog.

**13. Click the Save button.**



The window is saved in the built-in procedure window as a window recreation macro and is killed.

**14. Open the Procedure window, scroll up and read the window recreation macro.**

Note that the `NewWindowXOP1Window` command is included in the panel recreation macro.

**15. Close the Procedure window.**

**16. Choose Windows→Panel Macros→Panel0.**

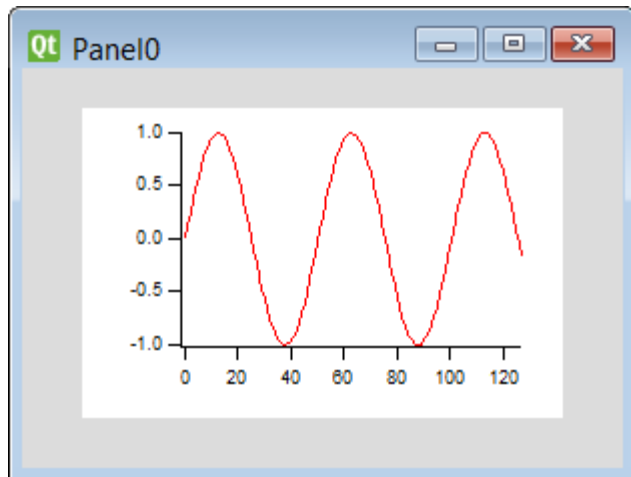
The panel window is recreated, including the XOP subwindow.

This concludes the Guided Tour of WindowXOP1.

## The Architecture of XOP Windows

An XOP window is an Igor window with an XOP container. To understand this you need to understand what we mean by "container".

Consider this Igor panel with a graph subwindow:



The graph subwindow is an Igor container that is embedded in the window. More precisely, the graph subwindow is an Igor container that is embedded in a panel container that is embedded in a window. Thus the hierarchy is:

```
Igor window
  Panel container
    Graph container
```

A container is a visible area of a window backed by an internal Igor object which manages it.

Each Igor window has one root container. The window's type and name are, by definition, the root container's type and name. This Igor window is considered a panel window because the root container is a panel container.

A container is a subwindow if it is not the root container for its window.

Some types of containers can act as hosts for other containers. Panel, graph and page layout containers can act as subwindow hosts. Tables, notebooks and XOP containers can not act as subwindow hosts. Any container that is capable of acting as a host can host multiple subwindows. In fact they can host a hierarchy of subwindows.

### Relationship to Qt Widgets

Igor7 is implemented on top of the Qt framework.

Qt has the concept of a widget, represented by the `QWidget` class. An Igor window is a Qt widget.

In SDI ("single document interface") mode on Macintosh a window is a top-level widget. In MDI ("multiple document interface") mode on Windows, a window is a `QMdiSubwindow` widget which is a child widget of the `QMdiArea` host window. A window widget in both SDI and MDI can host a hierarchy of child widgets.

For historical reasons, most Igor containers are not Qt widgets. So in the picture shown above, the window is a widget but the panel container and the graph container are not widgets. Rather they are drawn by Igor in the window. However, for reasons explained below, XOP containers *are* widgets.

Each operating system has its own platform-specific container representing screen content. On Windows this container is called a "window" and is referenced programmatically using a handle called an `HWND`. On Macintosh the container is called a "view" and is referenced programmatically using a pointer to an `NSView` object or an object derived from `NSView`. Both windows and views can be nested in a hierarchy. For example, a button is a child window or view in a parent window or view.

Because of different and overlapping terminology, explaining the relationship of all of these objects is challenging. For economy, in the following discussion we use the term "OS container" to refer to a Windows window or a Macintosh view. As you read it remember that "OS container" means "HWND on Windows and `NSView` on Macintosh".

We also use the terms "Igor container" to refer to any Igor container and "XOP container" to refer to an Igor container created and managed by an XOP. An Igor container is an Igor concept implemented by code in Igor. It is not the same as an OS container which is a concept implemented by code in the OS.

A `QWidget` can have an associated OS container. Most Igor windows have `QWidgets` and therefore have OS containers. With this in mind we can refine our schematic outline of the window pictured above as follows:

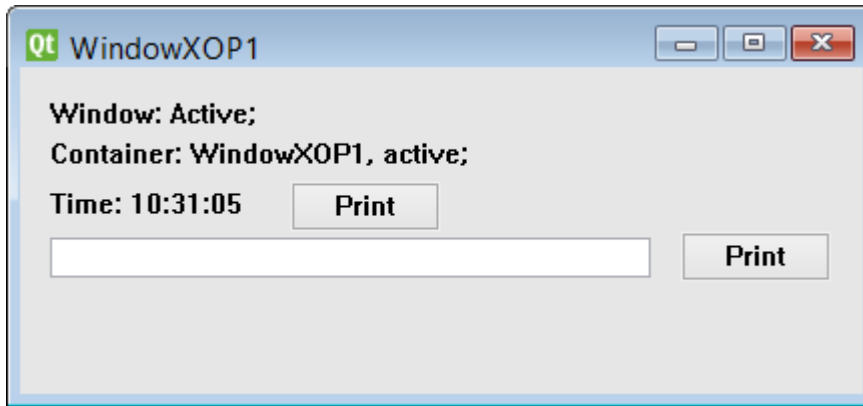
```
Igor window : QWidget(OS container)
    Panel container
    Graph container
```

This means that the Igor window is based on a `QWidget` and the widget has an OS container. The panel and graph containers do not have their own widgets and consequently do not have their own OS containers.

Things are different with XOP containers. We wanted XOP programmers to be able to use platform-specific code to implement XOP windows so that they could use pre-existing platform-specific code and platform-specific libraries as well as platform-specific programming techniques which they may already know. In order to achieve this, XOP containers must have associated OS containers which means that they must have associated widgets. Stated more concisely, unlike most Igor containers, XOP containers have widgets which have OS containers. Gizmo containers also must have OS containers because OpenGL requires it. So Gizmo containers and XOP containers are similar in this regard.

### XOP Window and Container Hierarchy

With that background established, we can now explain the hierarchy of XOP windows and containers. Consider this XOP window:



Here is the schematic outline:

```
Igor window : QWidget(OS container)
  XOP container : QWidget(OS container)
```

In other words, the Igor window has an OS container and the XOP container has an OS container that is a child of the window's OS container.

But it is a bit more complicated than that. The OS container associated with the XOP container's widget is owned by Qt. Thus Qt creates and manages that OS container and receives all events for it. What we need is an OS container that the XOP can manage. To achieve this, the XOP creates its own OS container and embeds it as a child of the XOP container's OS container. Thus the schematic becomes:

```
Igor window : QWidget(OS container)
  XOP container : QWidget(OS container)
    XOP OS container
```

The XOP OS container is created by the XOP and added by the XOP as a child of the XOP container's OS container. The XOP receives events for the XOP OS container from the OS and otherwise manages it. When we use the term "XOP OS container" we are referring to this XOP-created OS container, not to the Qt-created OS container.

As you can see, the architecture is rather complex. You may need to reread this section a few times before it all sinks in. But the key point is that your XOP creates and manages a platform-specific container (HWND on Windows, NSView on Macintosh) to receive events from the OS and to create your user interface.

By contrast, the interface to Igor is platform-independent, implemented via messages that Igor sends to your XOPEntry routine and by callbacks that you make to Igor.

## Source Code Organization

Open the WindowXOP1 folder in the Finder or Windows Explorer as we will refer to it in this section.

There is a subfolder for each development system containing the project files.

There is a Mac folder containing Macintosh-only source files and a Windows folder containing Windows-only source files. The Macintosh files are .mm (Objective-C++) files while the Windows files are .cpp (C++) files.

The main WindowXOP1 folder contains the platform-independent source code files. These fall into two categories. The files whose names start with "XOP" are intended to be application-independent. You will probably not have to modify them. The files whose names begin with "WindowXOP1" are application-specific. You will have to modify them for your purposes.

Once you clone the WindowXOP1 project, "WindowXOP1" in the file names will be changed to the name of your project. But for now we will inspect the WindowXOP1 files.

The following sections refer to WindowXOP1 source code. Open the WindowXOP1 project in your development system so you can look at specific code as needed to follow the discussion.

The Macintosh project includes the Windows files for ease of browsing and editing but the Windows files are excluded from compilation on Macintosh. The reverse is true on Windows. Consequently you can view all files in either project.

## Communication With the Operating System

The operating system sends platform-specific events, such as click, key and paint events, to your XOP code.

On Macintosh the OS sends events by calling routines in WindowXOP1View.mm which implements the WindowXOP1View class which is derived from the Cocoa NSView class. The WindowXOP1View class overrides NSView methods such as mouseDown and keyDown. Some of these methods call SendContainerNSEventToIgor so that Igor can respond to events in order to implement Igor-specific behaviors.

On Windows, you create a standard window procedure for your OS container. This is in the WindowXOP1HWND.cpp file and is named XOPWindowProc. The Windows OS sends messages like WM\_MOUSEBUTTONDOWN and WM\_KEYDOWN to XOPWindowProc. XOPWindowProc passes some of these messages to Igor by calling SendContainerHWNDEventToIgor so that Igor can respond to events in order to implement Igor-specific behaviors.

## Communication With Igor

While communication with the operating system is platform-dependent, communication with Igor is platform-independent.

Igor sends two types of messages specifically for window XOPs - window messages and container messages. Igor sends these messages to the XOP through the XOPEntry function, the same as it sends other messages.

Window messages apply to the window as a whole. Examples are kXOPWindowMessageActivate and kXOPWindowMessageResized. The window messages are defined in XOPWindows.h which is in the XOP-Support folder. Open that file and peruse the message definitions to get a sense of the type of window messages Igor sends to the XOP.

Container messages apply to an XOP container which may be a root container in an XOP window or a sub-window container in an Igor window such as a panel. Examples are kXOPContainerMessageActivate and kXOPContainerMessageResized. The container messages are defined in XOPContainers.h which is in the XOPSupport folder. Open that file and peruse the message definitions to get a sense of the type of container messages Igor sends to the XOP.

The XOPEntry routine calls HandleWindowMessage, in XOPWindowMessages.cpp, and HandleContainerMessage, in XOPContainerMessage.cpp. These functions then call the XOP routine needed to handle a given message.

Notice that there is a kXOPContainerMessageMouseClicked message. Earlier we said that click events are sent by the operating system directly to the XOP. This is true as noted in the comment for kXOPContainerMessageMouseClicked in XOPContainerMessages.cpp. kXOPContainerMessageMouseClicked is a message that Igor *would* send to the XOP except for the fact that the OS sends it directly to the XOP, circumventing Igor. There are other such messages as noted in the comments.

The HandleWindowMessage function and HandleContainerMessage function are generic and should work for any application. They call methods of classes defined by the XOP, described in the next two sections, to do the actual work. Consequently you should not need to deal directly with window or container messages nor should you need to modify the XOPWindowMessages.cpp or XOPContainerMessage.cpp files.

## XOPWindowProxy Class and WindowXOP1Window Class

For each XOP window the XOP creates an object of class `WindowXOP1Window`, which is derived from class `XOPWindowProxy`. The `XOPWindowProxy` class provides default behaviors that may be sufficient for a given application. The `WindowXOP1Window` class overrides `XOPWindowProxy` methods where necessary to provide application-specific behavior. You should not need to modify the `XOPWindowProxy` class. Rather you should modify the `WindowXOP1Window` class as necessary.

For example, when Igor sends the `kXOPWindowGetPreferredRect` message, which is used in conjunction with the `Windows→Control→Move to Preferred Position` menu command, the code in `XOPWindowMessages.cpp` calls the `getPreferredRect` method. This method is defined in the `XOPWindowProxy` class but overridden in the `WindowXOP1Window` class to provide application-specific behavior. Consequently, `WindowXOP1Window::getPreferredRect` is called which returns the preferred rectangle.

To see the sequence of calls you can set a breakpoint in the `getPreferredRect` method in `WindowXOP1Window.cpp`, run `WindowXOP1` under the debugger, and choose `Windows→Control→Move to Preferred Position`. Then examine the call stack in the debugger.

Most of the application-specific action is at the container level, not at the window level. Consequently most of the window behavior is provided by the generic `XOPWindowProxy` class and few methods need be overridden in the `WindowXOP1Window` class.

## XOPContainerProxy Class and WindowXOP1Container Class

For each XOP container the XOP creates an object of class `WindowXOP1Container`, which is derived from class `XOPContainerProxy`. The `XOPContainerProxy` class provides default behaviors that may be sufficient for a given application. The `WindowXOP1Container` class overrides `XOPContainerProxy` methods where necessary to provide application-specific behavior. You should not need to modify the `XOPContainerProxy` class. Rather you should modify the `WindowXOP1Container` class as necessary.

For example, Igor sends the `kXOPContainerUpdateEditMenu` message when the user clicks the Edit menu while an XOP container is active. The code in `XOPContainerMessages.cpp` calls the `updateEditMenuItems` method. This method is defined in the `XOPContainerProxy` class but overridden in the `WindowXOP1Container` class to provide application-specific behavior. Consequently, `WindowXOP1Container::updateEditMenuItems` is called which enables items in the Edit menu as appropriate.

To see the sequence of calls you can set a breakpoint in the `updateEditMenuItems` method in `WindowXOP1Container.cpp`, run `WindowXOP1` under the debugger, and click the Edit menu. Then examine the call stack in the debugger.

Most of the methods defined in the `XOPContainerProxy` need to be overridden by the `WindowXOP1Container` to provide application-specific behavior.

## Platform-Independent Source Files

This section provides a brief discussion of the main platform-independent source files.

### WindowXOP1.cpp

`WindowXOP1.cpp` contains the XOPs main function, named `XOPMain`. `XOPMain` creates the initial `WindowXOP1` window at launch time.

Other functionality includes:

- Dispatching messages from Igor via the `XOPEntry` function
- Handling window-independent and container-independent menu items

### WindowXOP1Window.cpp

WindowXOP1Window.cpp contains routines for creating and killing windows and also implements the WindowXOP1Window class. This class derives from the XOPWindowProxy class and overrides few of its methods.

The CreateWindowXOP1Window routine creates a new WindowXOP1 window and its root container. It also creates the WindowXOP1Window class object that is associated with the new window. A pointer to this object is registered with Igor. The XOP retrieves the pointer from Igor whenever it needs to access the object.

The KillWindowXOP1Window routine calls back to Igor to kill the window. XOP code calls it if it wants to initiate the killing of the window. Igor can also initiate the killing of a windows, such as when the user executes a KillWindow command. However the killing is initiated, Igor sends the kXOPWindowMessageWindowBeingKilled message to the XOP. At that time, the XOP kills the WindowXOP1Window class object and any associated data. This complex process is explained in the comments for KillXOPWindow in XOPSupport/XOPWindows.c. A good way to see how this works is to set a breakpoint in the WindowXOP1Window destructor and then create and kill a WindowXOP1 window.

### WindowXOP1Container.cpp

WindowXOP1Container.cpp contains routines for creating and killing XOP containers and also implements the WindowXOP1Container class. This class derives from the XOPContainerProxy class and overrides many of its methods.

The CreateWindowXOP1Container routine creates a new WindowXOP1 container. Indirectly, through the platform-specific AddXOPNSView or AddXOPHWND routines, it creates the WindowXOP1Container class object that is associated with the new container. A pointer to this object is registered with Igor. The XOP retrieves the pointer from Igor whenever it needs to access the object.

The killing of an XOP container is always initiated by Igor. This occurs when the container's parent is killed. At that time Igor sends the kXOPContainerMessageContainerBeingKilled message to the XOP and the XOP kills any data associated with the container. This process is explained in the comments for KillXOPContainer in XOPSupport/XOPContainers.c. A good way to see how this works is to set a breakpoint in the WindowXOP1Container destructor and then create and kill a WindowXOP1 window.

### WindowXOP1RecreationMacros.cpp

WindowXOP1RecreationMacros.cpp implements the generateXOPContainerRecreationMacroText method of the WindowXOP1Window class. It is in a separate file because it tends to be long and complex and would clutter up WindowXOP1Window.cpp. The generateXOPContainerRecreationMacroText method is called when Igor generates a recreation macro for an XOP target window or for an Igor window with an XOP container.

### WindowXOP1Functions.cpp

WindowXOP1Functions.cpp implements external functions that WaveMetrics uses for testing purposes. Your XOP does not need these functions.

### WindowXOP1Operations.cpp

WindowXOP1Operations.cpp implements the NewWindowXOP1Window command line operation.

If your XOP will implement multiple operations we recommend creating one file per operation for easier browsing and editing.

The ExecuteNewWindowXOP1Window function implements the typical flags and features of an Igor operation that creates a new window.

## Macintosh XOP Window Programming

On Macintosh all event handling, user interface creation and drawing for XOP containers is done in Cocoa which uses Objective-C (.m files) or Objective-C++ (.mm files). WindowXOP1 uses Objective-C++.

In order to understand the Macintosh code you need to understand, at a bare minimum, the basics of Objective-C and Cocoa. If you have no experience with these technologies, you will have to bring yourself up-to-speed. The learning curve is steep. Here are some resources for getting started:

**Start Developing Mac Apps Today** (Apple online tutorial)

**Programming with Objective-C** (Apple introduction to Objective-C)

**Mac App Programming Guide**

On Macintosh the XOP OS container is an object of class XOPView which is derived from the Cocoa NSView class. The Cocoa system calls methods of the NSView class to send events to the program. The XOPView class overrides these methods. For some method calls the method must call SendContainerNSEventToIgor to allow Igor to respond to the event and implement Igor-specific behaviors.

Following the Cocoa custom, the XOPView object represents the visible aspects of the view while an XOPViewController object handles the logic and interaction with other parts of the system. The XOPController class handles the overall logic for the XOP as a whole including creation and destruction of the other objects.

The WindowXOP1 code does not utilize Cocoa's Automatic Reference Counting (ARC) because ARC is not supported for 32-bit programs. Instead it implements the old-fashioned manual reference counting which is tedious and error prone.

Most users will have no need to run the 32-bit version of Igor7 but instead will run the 64-bit version. However some users may be forced to run the 32-bit version, for example, because they rely on an XOP that is not available in 64 bits. This may occur because the XOP programmer who wrote the XOP is no longer available or does not have time to port it to 64 bits. For these reason WaveMetrics continues to provide a 32-bit version of Igor7.

If you do not need to create a 32-bit version of your XOP then you can change the project to use ARC. See Apple's **Transitioning to ARC Release Notes** for details.

## Macintosh Source Files

This section provides a brief discussion of the main Macintosh source files.

### WindowXOP1Controller.mm

WindowXOP1Controller.mm implements the AddXOPNSView and KillXOPNSView functions.

The AddXOPNSView function is called from WindowXOP1CreateContainer. It loads the nib which causes Cocoa to create the XOPViewController and XOPView objects. It then adds the XOP view as a child of the XOP container view. This is a complex and confusing process which is documented in comments in the file. Fortunately it should work with minor modifications for your XOP.

AddXOPNSView also creates the WindowXOP1Container object which represents the container in the XOP code.

### WindowXOP1ViewController.mm

The WindowXOP1ViewController.mm file implements the actions of the XOP view. Specifically it provides action procedures for the buttons.

### WindowXOP1View.mm

The WindowXOP1View.mm file implements the XOPView object which derives from the Cocoa NSView class. XOPView overrides many NSView functions, such as mouseDown and keyDown. Cocoa calls these methods to pass events to the view. The XOPView class also overrides the drawRect method which Cocoa calls anytime the view needs to be repainted.

## Windows XOP Window Programming

On Windows all event handling, user interface creation and drawing for XOP containers is done in using the Windows API. The part of the API that deals with windows is documented at MSDN under Desktop Technologies:

### Windows and Messages

As for any Windows API window, the program must register a window class and then create an instance of the class. It must provide a window procedure function to which Windows sends messages when events happen. For some messages the window procedure must call SendContainerHWNDEventToIgor to allow Igor to respond to the event and implement Igor-specific behaviors.

## Window Source Files

This section provides a brief discussion of the main Windows source files.

### WindowXOP1HWND.cpp

WindowXOP1HWND.cpp contains the crux of the Windows implementation. The first time an XOP container is created it registers the WindowXOP1 window class by calling RegisterClassEx. The file also contains the XOPWindowProc function which is the window procedure function to which the Windows OS sends messages.

The AddXOPHWND function is called from WindowXOP1CreateContainer. It creates the XOP OS container and adds it to the XOP container as a child window. It also creates the WindowXOP1Container object which represents the container in the XOP code.

### WindowXOP1CommandsWin.cpp

WindowXOP1CommandsWin.cpp handles commands sent by Windows when the user clicks a button. Such commands are sent to the XOPWindowProc function via WM\_COMMAND messages.

### WindowXOP1DrawingWin.cpp

WindowXOP1DrawingWin.cpp draws the XOP container visible contents. However child windows such as buttons draw themselves.

## Menu Handling for XOP Windows and Containers

Because Igor controls the main menu bar and all of its menus, all interaction with menus occurs via messages that Igor sends to the XOP, not through platform-specific techniques.

### Enabling Menu Items

Igor sends different messages depending on the nature of the menus or menu items that need to be updated:

The **XOPMENUENABLE** message tells the XOP to update menu items that are independent of the active window or container. These are menu items that the XOP added to Igor menus or to top-level XOP-created menus. See UpdateIndependentXOPMenuItems in WindowXOP1.cpp.

Igor sends the **kXOPWindowMessageUpdateFileMenu** message when the File menu needs to be updated and your XOP window is the active window. See the XOPWindowProxy::updateFileMenuItems method.



Igor sends the **kXOPWindowMessageUpdateEditMenu** message when the Edit menu needs to be updated and your XOP window is the active window. See the `XOPWindowProxy::updateWindowEditMenuItems` method. It is unlikely that you will need to modify this because most of the items in the Edit menu are container-oriented rather than window-oriented.

Igor sends the **kXOPWindowMessageUpdateWindowsMenu** message when the Windows menu needs to be updated and your XOP window is the active window. See the `XOPWindowProxy::updateWindowsMenuItems` method. It is unlikely that you will need to modify this because the default implementation is sufficient.

Igor sends the **kXOPContainerMessageUpdateEditMenu** message when the Edit menu needs to be updated and your XOP container is the active container. See the `WindowXOP1Container::updateEditMenuItems` method. Most Edit menu items are enabled by this method.

Igor sends the **kXOPContainerMessageUpdateSlotMenu** message when the slot menu needs to be updated and your XOP container's slot menu is visible. The slot menu is the menu that changes based on the type of window that is active. When a graph is active, the Graph menu is the slot menu and when a table is active, the Table menu is the slot menu. An XOP declares a slot menu using the STR# 1160 resource. See `WindowXOP1.r` and `WindowXOP1WinCustom.rc` for examples. If an XOP declares a slot menu then Igor displays it as appropriate and calls `WindowXOP1Container::updateSlotMenuItems` to update its items.

Igor7 sets the Windows→Close menu item to Close or Hide based on the XOP's response to the `kXOPWindowMessageCloseMeansHide` message. If the XOP responds with true, Igor sets the menu item to "Hide". Otherwise it sets it to "Close". In both cases Igor enables the menu item. The code in `XOPWindowMessages.cpp` calls `XOPWindowProxy::closeMeansHide` which returns true for non-target windows and false for target windows. If you want to change this, add a `closeMeansHide` method to the `WindowXOP1Window` class.

## Responding to Menu Selections

If the user chooses a built-in menu item that applies to the active window while your XOP window is active, Igor sends a message specific to that menu item. The generic routines in `XOPWindowMessages.cpp` field the message and call the appropriate `WindowXOP1Window` method.

For example, if the user chooses Windows→Control→Move to Preferred Position, the generic routines call `XOPWindowProxy::moveToPreferredPosition` which implements the default behavior. If you wanted to override the default behavior you would add a `moveToPreferredPosition` method to the `WindowXOP1Window` class.

If the user chooses a built-in menu item that applies to the active container while your XOP container is active, Igor sends a message specific to that menu item. The generic routines in `XOPContainerMessages.cpp` field the message and call the appropriate `WindowXOP1Container` method.

For example, if the user chooses Edit→Cut, the generic routines call `WindowXOP1Container::cut` which implements the cut operation.

If the user chooses a menu item added by the XOP, whether the item was added to an Igor menu or is part of an XOP menu, Igor sends the `MENUIITEM` message which is described under **Responding to Menu Selections** on page 150. The `XOPEntry` routine calls `XOPMenuItem` in `WindowXOP1.cpp`.

## Cursor Handling for XOP Windows and Containers

Cursor handling is different on Macintosh and Windows.

### Macintosh Cursor Handling

The normal way to handle the cursor in Cocoa is for each `NSView` to control the cursor when the mouse is over it. This is described in Apple's **Cocoa Event Handling Guide** under **Using Tracking-Area Objects**.

Unfortunately Qt, on which Igor7 is based, interferes with this approach. Qt traps mouse moved events and sets the cursor before the normal Cocoa mechanism kicks in. It also sets the cursor at other times, such as when the window is activated. This defeats any cursor-setting implemented by an `NSView`.

After much investigation we concluded that there is no practical way to wrest control of the cursor from Qt on Macintosh. Therefore the only solution is to tell Qt which cursor to use by calling the XOP Toolkit callbacks `ArrowCursor`, `IBeamCursor`, `WatchCursor`, `HandCursor` or `SetXOPContainerMouseCursor`. On Macintosh these routines tell the Qt widget in which the XOP `NSView` lives which cursor to use. Qt sets the cursor when the mouse is moved over the widget.

The cursor codes for `SetXOPContainerMouseCursor` are listed in the `IgorMouseCursorCode` enum in `IgorXOP.h`. To see what each cursor looks like, run the "Mouse Cursor Control.pxp" example experiment that ships with Igor7.

When the user moves the mouse over an XOP container, Igor sends the `kXOPContainerMessageMouseMove` message to the XOP. This causes `WindowXOP1Container::mouseMoved` to be called. It calls the platform-dependent `HandleMouseMoveMoved` routine which calls `SetXOPContainerMouseCursor`.

### Windows Cursor Handling

Unlike on Macintosh, the XOP is able to wrest control from Qt because the XOP window proc is called instead of the Qt window proc. Therefore cursor handling on Windows is done using standard Windows API practices in response to the `WM_MOUSEMOVE` message. You can use the XOPSupport routines `ArrowCursor`, `IBeamCursor`, `WatchCursor`, `HandCursor` and `SetXOPContainerMouseCursor` or Windows platform-dependent techniques instead. This is illustrated in the `XOPWindowProc` function in `WindowXOP1HWND.cpp`.

On Windows `ArrowCursor`, `IBeamCursor`, `WatchCursor`, `HandCursor` and `SetXOPContainerMouseCursor` call the Windows `SetCursor` function directly. Unlike Macintosh, on Windows `WindowXOP1Container::mouseMoved` and `HandleMouseMoveMoved` are not called.

The cursor codes for `SetXOPContainerMouseCursor` are listed in the `IgorMouseCursorCode` enum in `IgorXOP.h`. To see what each cursor looks like, run the "Mouse Cursor Control.pxp" example experiment that ships with Igor7.

## Activating, Showing and Hiding Windows

Normally the showing and hiding of windows is left in the hands of the user. If you want to programmatically show a window, call **ShowIgorWindow** or **ShowAndActivateIgorWindow**. If you want to programmatically hide a window, call **HideIgorWindow** or **HideAndDeactivateIgorWindow**. These functions are documented in `XOPSupport/XOPWindows.cpp`.

## Messages for XOP Windows

The window-related messages that Igor sends to XOPs are defined in `XOPSupport/XOPWindows.h`. These messages are entirely handled in the file `XOPWindowMessages.cpp` which is part of the `WindowXOP1` XOP. That file includes detailed comments explaining the messages. However you never need to deal with window messages directly as the routines in `XOPWindowMessages.cpp` handle them by calling the appropriate method of the `XOPWindowProxy` class or the `WindowXOP1Window` class.

**NOTE:** The window messages are documented in `XOPWindowMessages.cpp`, not in this manual.

You should not need to modify `XOPWindowProxy.h` or `XOPWindowProxy.cpp`. Instead, if you want to customize the response to window messages, modify `WindowXOP1Window.h` and `WindowXOP1Window.cpp`. This is where you override methods in the `XOPWindowProxy` class to provide customized behavior.

## Messages for XOP Containers

The container-related messages that Igor sends to XOPs are defined in XOPSupport/XOPContainers.h. These messages are entirely handled in the file XOPContainerMessages.cpp which is part of the WindowXOP1 XOP. That file includes detailed comments explaining the messages. However you never need to deal with container messages directly as the routines in XOPContainerMessages.cpp handle them by calling the appropriate method of the XOPContainerProxy class or the WindowXOP1Container class.

**NOTE:** The container messages are documented in XOPContainerMessages.cpp, not in this manual.

You should not need to modify XOPContainerProxy.h or XOPContainerProxy.cpp. Instead, if you want to customize the response to container messages, modify WindowXOP1Container.h and WindowXOP1Container.cpp. This is where you override methods in the XOPContainerProxy class to provide customized behavior.

## Igor Window Coordinates

This section describes a platform-independent coordinate system called "Igor coordinates". Igor coordinates are most-commonly used in command line operations such as:

```
Display/W=(10,45,410,245) jack
```

A command like this should have roughly the same effect on any platform. Consequently we need a platform-independent coordinate system.

Some XOPSupport routines require Igor coordinates while others require system-dependent pixel coordinates. The documentation for a given XOPSupport routine specifies which coordinate system it requires.

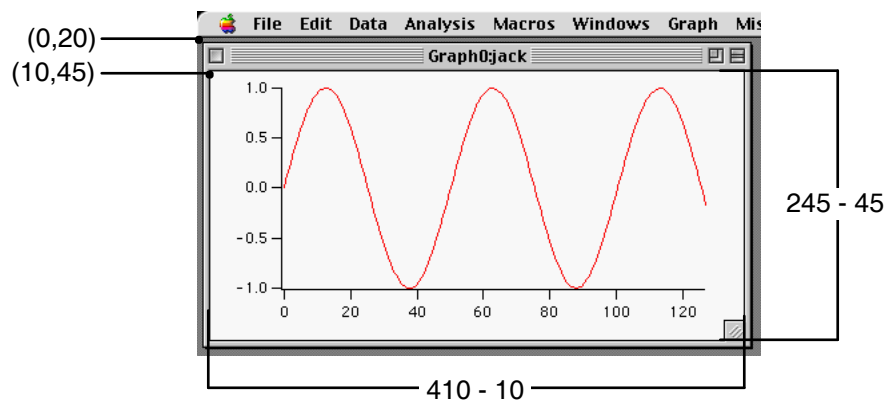
Igor's coordinate system originated at the birth of Igor in the spring of 1987. It was based on the Mac OS coordinate system of the time. The operating system was Mac OS 4.1. Igor still uses the same coordinate system which has been adapted for Windows and for high-resolution screens.

Igor uses global coordinates in units of points, relative to the top/left corner of the main screen. On a normal-resolution Macintosh a point is assumed to be equivalent to one pixel. On a Retina Macintosh a point is assumed to be equivalent to two pixels both vertically and horizontally for a total of four square pixels.

Because the Macintosh menu bar takes 20 points, the coordinate of the top edge of the desktop area, which corresponds to the bottom edge of the menu bar, is 20.

When specifying a window size and position, you provide the top/left coordinate of the window's "content region". This is the area of the window exclusive of the title bar and frame. The following picture shows a graph window created on Macintosh using the command

```
Display/W=(10,45,410,245) jack
```



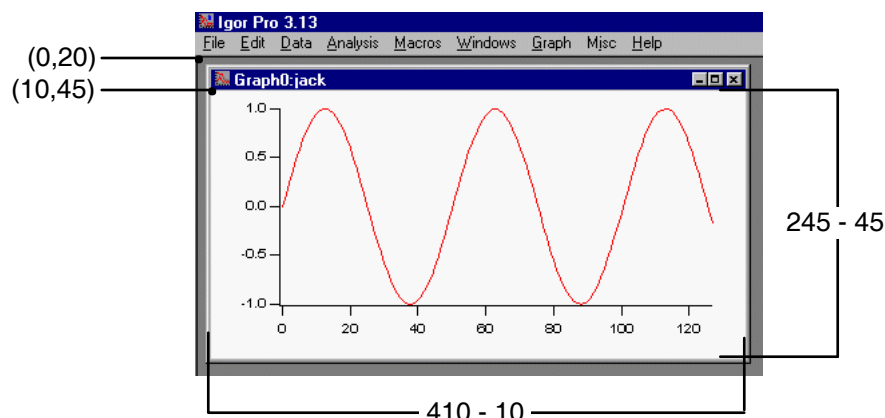
## Chapter 9 — Adding Windows

The vertical coordinate 45 takes the height of the menu bar and the heights of the window frame and title bar into account.

For compatibility on Windows the vertical coordinate 20 is taken to be the top of the desktop, which corresponds the bottom of the menu bar, as it does on Macintosh.

The following picture shows a graph window created on Windows using the command

```
Display/W=(10,45,410,245) jack
```



On Windows the correspondence between points and pixels is set by the Display control panel. Igor coordinates are stored as floating point because, to accurately reposition a window, we need to use fractional points in /W=(left,top,right,bottom) flags.

The **TransformWindowCoordinates** XOPSupport routine converts between native Macintosh or native Windows coordinates and Igor coordinates.

To get your window's size and position in Igor coordinates, call the **GetIgorWindowPositionAndState** 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 **SetIgorWindowPositionAndState** XOPSupport routine. These routines work the same on Macintosh and Windows.

Window-related XOPSupport routines are documented in XOPSupport/XOPWindows.cpp.

## 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 such as MoveWindow, GetWindow, SetWindow and KillWindow. The name of a window is by definition the name of its top-level container.

Target windows are tied to the current experiment. Igor kills them when the experiment is closed via New Experiment or Open Experiment. Non-target windows, like the command window and Data Browser, are independent of the current experiment and persist across experiments.

A target window can be stored as a window recreation macro and recreated later by running the recreation macro. When an experiment is saved Igor saves a window recreation macro for each target window. When an experiment is opened Igor runs the window recreation macro to recreate the window. Igor does not create recreation macros for non-target windows.

Igor Pro 6 supported five built-in target window types: graphs, tables, page layouts, notebooks, and control panels. The Gizmo window type was added via an XOP. In Igor Pro 7 the Gizmo window was converted from an XOP to a built-in type.

An XOP can add a target window type to Igor. 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.

An XOP target window can be embedded in an Igor window as a subwindow.

The WindowXOP1 XOP creates a target window when you execute the `NewWindowXOP1Window` command. It creates a subwindow when you execute `NewWindowXOP1Window/HOST=<host-spec>`. It does this by using the `kCreateXOPWindowAsTargetWindow` option when calling `CreateXOPWindow`.

If you omit the `kCreateXOPWindowAsTargetWindow` option, `CreateXOPWindow` creates the window as a non-target window.

## Window Recreation Macros

Igor sends the `kXOPContainerMessageRecreationText` message when it generates a window recreation macro for an XOP window or for an Igor window with an XOP container as a subwindow. Code in `XOP-ContainerMessages.cpp` calls `WindowXOP1Container::generatXOPContainerRecreationText` which returns the required recreation commands. Because recreation macro generation can require a lot of code, to reduce clutter in `WindowXOP1Container.cpp`, `WindowXOP1Container::generatXOPContainerRecreationText` is implemented in a separate file - `WindowXOP1RecreationMacros.cpp`.

## Window Hook Functions

User-defined window hook functions, as described under "Window Hook Functions" in the Igor help, are supported for XOP target windows except for the following messages:

Event Code	Event Name	Notes
7	<code>cursormoved</code>	Not applicable
8	<code>modified</code>	Not supported - This would require a callback from the XOP to Igor anytime the XOP content was modified which would be time-consuming
14	<code>subwindowKill</code>	Not applicable - XOP windows don't support subwindows
18	<code>showTools</code>	Not applicable
19	<code>hideTools</code>	Not applicable
20	<code>showInfo</code>	Not applicable
21	<code>hideInfo</code>	Not applicable
23	<code>spinUpdate</code>	Not applicable

## XOP Window Issues

Here are the main known issues with XOP windows in Igor7:

- An XOP can define only one slot menu

Here are the main unimplemented features for XOP windows in Igor7:

- Edit→Duplicate Window does nothing for an XOP window.

## XOP Subwindow Issues

You can add an XOP container as a subwindow to an Igor graph, page layout or panel.

Here are the main known issues with XOP subwindows in Igor7:

- XOP subwindows are always displayed on top of graph, table and notebook subwindows.

- In subwindow layout mode (show the drawing tools, click the drawing icon, click the XOP subwindow, click the thick frame around the subwindow), if you press Cmd-T or Ctrl-T, the tools are not hidden.
- There is no way to export the XOP subwindow when exporting a graph or page layout window.
- There is no way to print the subwindow when printing a graph or page layout window.
- When you change the magnification of the graph, the magnification of the XOP subwindow does not change.

Here are the main unimplemented features for XOP subwindows in Igor7:

- Subwindow frames are not supported.
- There is no way to add the name of a type of XOP subwindow to the New submenu that you get when you right-click a host container in drawing mode.
- There is no support for drawing XOP subwindow at different resolutions. This is needed for several purposes:
  1. To draw an XOP subwindow in a layout at other than 100%.
  2. To draw an XOP subwindow in a graph whose resolution is other than 100% (e.g., `ModifyGraph expand=2`).
  3. To draw an XOP subwindow in a panel whose resolution is other than 100% (e.g., `SetIgorOption PanelResolution=144`).
  4. To export a panel, graph or layout containing an XOP subwindow at a resolution other than 100%.

## Changes to XOP Window Programming in Igor Pro 7

This section discusses changes required for existing XOPs that add windows to Igor. You need to read this section only if you have an existing Igor6 window XOP and you want to port your XOP to Igor7. For an overview of the process of adding windows see **Adding Windows** on page 155.

In Igor6 and before Igor was implemented on top of the Carbon Framework on Macintosh and on top of the Windows API on Windows. Consequently the XOP window protocol was platform-specific, using the Carbon or Windows API depending on the platform.

With Igor7 WaveMetrics re-implemented Igor based on Qt - a platform-independent framework for application development. In order for XOP windows to integrate nicely into Igor7, WaveMetrics has created a new XOP window protocol. The protocol is implemented by Igor7 and XOP Toolkit 7.

**NOTE:** Igor6 window XOPs are not compatible with Igor7. To run your window XOP with Igor7 you must revise it using XOP Toolkit 7 to use the Igor7 window XOP protocol.

**NOTE:** Igor7 window XOPs are not compatible with Igor6. They require Igor7 or later.

The Igor7 window protocol is completely different from the Igor6 protocol. Because of this, to port an existing window XOP to Igor7 we recommend that you start with the `WindowXOP1` sample XOP, clone it using the "Create New XOP Project.pxp" experiment file, and then add your custom code from your previous window XOP to the new one.

The main changes in the Igor7 XOP window protocol are:

- Igor creates the window instead of the XOP
- You use the type `IgorWindowRef` as a pointer to the window structure instead of `XOP_WINDOW_REF`
- A new concept, the container, is introduced
- You can embed an XOP container in an Igor window as a subwindow

- All of the old window-related messages (e.g., CLICK, KEY) from Igor are no longer supported, replaced by new messages (e.g., kXOPContainerMessageMouseClicked, kXOPContainerMessageKeyPressed)

## IgorWindowRef Data Type Replaces XOP\_WINDOW\_REF

In Igor6 and XOP Toolkit 6 you used the type XOP\_WINDOW\_REF to refer to the structure describing an XOP window. An XOP\_WINDOW\_REF is a Carbon WindowPtr on Macintosh and a Windows API HWND on Windows.

In Igor7 and XOP Toolkit 7 you use the type IgorWindowRef instead of XOP\_WINDOW\_REF. An IgorWindowRef is a pointer to a WaveMetrics object that describes an Igor window. You obtain an IgorWindowRef via a callback to Igor. The details of what an IgorWindowRef points to are private and may change from one version of Igor to the next. You use callbacks to operate on the window.

## TellIgorWindowStatus Removed

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.

In Igor Pro 6, to participate in these features, an XOP had to call the TellIgorWindowStatus callback. In Igor Pro 7, this is handled internally by Igor. Consequently the TellIgorWindowStatus callback was removed from XOP Toolkit 7. If you are porting an XOP to XOP Toolkit 7, remove all calls to TellIgorWindowStatus.

## Removed Window-Related XOPSupport Routines

The section list window-related routines removed from XOP Toolkit 7 and replacement routines, if any.

The replacement routines are documented in XOPSupport/XOPWindows.c, not in this manual.

Removed Routine	Replacement Routine
GetXOPWindow	None
GetActiveWindowRef	GetActiveIgorWindow
IsXOPWindowActive	IsIgorWindowActive
ShowXOPWindow	ShowIgorWindow
HideXOPWindow	HideIgorWindow
ShowAndActivateXOPWindow	ShowAndActivateIgorWindow
HideAndDeactivateXOPWindow	HideAndDeactivateIgorWindow
SetXOPWindowTitle	SetIgorWindowTitle
GetXOPWindowPositionAndState	GetIgorWindowPositionAndState
SetXOPWindowPositionAndState	SetIgorWindowPositionAndState
GetXOPWindowIgorPositionAndState	GetIgorWindowIgorPositionAndState
SetXOPWindowIgorPositionAndState	SetIgorWindowIgorPositionAndState
TellIgorWindowStatus	None
IBeamCursor	SetXOPContainerMouseCursor
HandCursor	SetXOPContainerMouseCursor
SpinCursor	SpinProcess

### TU ("Text Utility") Windows

A TU window is a "text utility" window - a simple plain text editing window. The VDT2 sample XOP illustrate the use of a TU window.

#### Changed Text Utility Window Callbacks

The Text Utility Window callbacks are documented in XOPSupport/XOPTextUtilityWindow.c, not in this manual. This section lists Text Utility Window routines that changed in XOP Toolkit 7.

The TUNew2 callback now takes an IgorWindowRef\* parameter.

The meaning of the second parameter to the TUFind callback has changed.

The TUClick callback now takes a WMMouseEventRecord\* parameter.

The TUNull callback now takes a WMMouseEventRecord\* parameter.

The TUKey callback now takes a WMKeyboardEventRecord\* parameter.

The TUSFWriteFile routine allFlag parameter is ignored by Igor7 which always writes the entire file, even if allFlag is 0.

The TUSetStatusArea function ignores the statusAreaWidth parameter when running with Igor7 or later.

### 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 67.



# 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 95.

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 117.

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 automatically multithreaded Igor operations like ImageInterpolate, 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 an automatically multithreaded Igor operation such as ImageInterpolate 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 from the main thread
- 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 **MDSetTextWavePoint-Value**.

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

With Igor Pro 6.20 or later, your XOP can do a callback to Igor from an Igor preemptive thread if the callback is designated as Igor-thread-safe. The documentation for each routine, in Chapter 15, **XOPSupport Routines**, tells you if the routine is Igor-thread-safe or 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 as described under **Checking Igor's Version** on page 70.

Even when 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.

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 `NULL` 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 177.)

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 182)	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 183)	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 184)	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 184)	Igor Thread	Main thread numeric waves	No transfer required
Method 4 (page 185)	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.

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 **Checking Igor's Version** on page 70, 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	<p>The thread worker function calls the external operation to start data acquisition</p> <p>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</p> <p>The external operation calls the <b>ThreadGroupPutDF</b> callback to transfer the data folder to the thread group's output queue</p> <p>The external operation returns to the thread worker function</p>
Igor sends the IDLE message to the XOP	
The XOP calls <b>ThreadGroupGetDF</b> to see if data from the preemptive thread is available and to transfer it to the main thread's data hierarchy	<p>If more data is to be acquired, the thread worker function repeats the process</p> <p>The thread worker function returns and the Igor preemptive thread stops</p>

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 starts 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
<hr/>	
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	
<hr/>	

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	<p>The thread worker function calls the external operation, passing the waves as parameters</p> <p>The external operation acquires data into the waves and marks them as modified (WaveHandleModified)</p> <p>The external operation returns to the thread worker function</p> <p>The thread worker function returns and the Igor preemptive thread stops</p>

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 177.

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 126.

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	

Igor Main Thread	XOP Private Thread
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 177 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 7 supports development of 32-bit and 64-bit XOPs for both Macintosh and Windows. The XOP-Support library and all of the sample XOPs are provided in both 32 bits and 64 bits.

As of Igor7, Igor is available as both 32-bit and 64-bit executables on both Macintosh and Windows. In Igor6, the 64-bit version was available on Windows only.

We refer to the 64-bit version as IGOR64. The 32-bit version is called IGOR32.

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 allows you 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, the operating system may impose further limits.

On Macintosh each process can theoretically access the full 17 billion GB. In 64-bit Windows operating systems, the limit depends on the operating system version as explained at:

[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.

Although most applications will never need the address space provided by IGOR64, 64-bits has become the defacto standard. Therefore, with Igor7 or later, we recommend that you use IGOR64. The only reason to use IGOR32 is if you use an XOP that has not yet been ported to 32 bits.

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.

IGOR64 loads only XOPs whose names end with "64" (e.g., WaveAccess64.xop). If your XOP's name ends with a digit (e.g., VDT2.xop), append "-64" (e.g., VDT2-64.xop).

IGOR32 loads only XOPs whose names do not end with "64" (e.g., WaveAccess.xop or VDT2.xop).

## 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 6 for details.

## Updating Your Existing Project for 64-bit Development

If you have an existing XOP project that you want to use to create a 64-bit version of your XOP, you must first update your project to add a 64-bit target.

To update an Xcode project, see **Xcode Project Changes For XOP Toolkit 7** on page 427.

To update an Visual C++ project, see **Visual C++ Project Changes For XOP Toolkit 7** on page 434.

Most code written for 32 bits will need some modification in order to successfully compile and run in 64 bits. The remaining sections of this chapter address the relevant issues.

## Data Types For 64-bit Compatibility

When compiling a 64-bit executable, the long data type is 32 bits on Windows but 64 bits on Macintosh. Consequently the type long is problematic when compiling for 64 bits. Therefore in XOP Toolkit 6, all longs were changed to ints, if a signed 32 bit value was sufficient, 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 the same size whether running in 32 or 64 bits. This is the case, for example, when reading or writing a field of a structure that is written to disk and read by both versions of an XOP.

The following data types are defined in XOPSupport/WMTypes.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)
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 32-bit code, 64 bits in 64-bit code Used for integers that may hold pointers and for other situations where an integer should hold 32 bits in 32-bit code and 64 bits in 64-bit code
BCInt	Byte-count integer 32 bits in 32-bit code, 64 bits in 64-bit code Used to hold a byte count that needs to be 32 bits in 32-bit code and 64 bits in 64-bit code
IndexInt	Index integer 32 bits in 32-bit code, 64 bits in 64-bit code Used to hold a value used to index a pointer Also used to hold wave point numbers
CountInt	Count integer 32 bits in 32-bit code, 64 bits in 64-bit code Used to hold an item count that needs to be 32 bits in 32-bit code and 64 bits in 64-bit code 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.

If you are porting an old XOP to 64 bits, you will need to modify your code as explained below. If you are working on a new XOP based on an XOP Toolkit 7 sample, the issues were already handled in the sample 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:

<https://developer.apple.com/library/mac/documentation/Darwin/Conceptual/64bitPorting>

### 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 = WMGetHandleSize(waveH);
```

In this case, the value being stored in len could exceed 32 bits so this warning should be fixed like this:

```
BCInt len = WMGetHandleSize(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 WMGetHandleSize to an int:

```
int len = (int)WMGetHandleSize(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 longs in your code.

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, the XOP Toolkit provides 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 188 for the complete list.

There are four situations with regard to longs:

- 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 introduced a type for this purpose: **XOPIORecResult**. **XOPIORecResult** is a typedef for **SInt32** in IGOR32 and **SInt64** in IGOR64.

If you are porting an old XOP to XOP Toolkit 7, 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** was changed from long to int. So, if you are porting an old XOP to XOP Toolkit 7, 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** was 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 219.

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 an alias or shortcut for MyXOP.xop in the "Igor Extensions" folder and an alias or 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.

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

### 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.

Modern compilers consider this form of main to be an error.

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 GCC 4.3 or later or LLVM
HOST_IMPORT int
main(IORecHandle ioRecHandle)
```

```
// New XOPMain - Requires Igor Pro 6.20 or later
HOST_IMPORT int
XOPMain(IORecHandle ioRecHandle)
```

main and XOPMain behave identically.

Because of compiler requirements, all new XOPs and all XOPs that are being updated must use XOPMain, not main. This means that these XOP will require Igor Pro 6.20 or later. Consequently, you must include a check of Igor's version in your XOPMain function as illustrated under **Checking Igor's Version** on page 70.

### WM Memory XOPSupport Routines

The WM memory XOPSupport routines provide a mechanism by which Igor and XOPs can exchange data in a compatible way.

WM memory XOPSupport routines use two data types, Ptr and Handle. They both provide access to a block of memory allocated in the heap. Ptr is rarely used while Handle is commonly used.

Handles are used by Igor to pass variable-length data to XOPs and to receive variable-length data from them. For example, Igor uses a Handle to pass a string parameter to an external function which returns a string result to Igor via a Handle.

The XOP Toolkit provides routines for dealing with Handles, such as WMNewHandle, WMGetHandleSize, and WMDisposeHandle. These routines evolved from the Macintosh Memory Manager which dates back to the 1980's. Igor and the XOP Toolkit still use Handles for backward compatibility.

## Chapter 12 — Other Programming Topics

---

Wave handles and data folder handles use different memory management techniques. Consequently the routines described in this section must not be used on wave handles or on data folder handles. Instead use the specific XOPSupport routines listed under **Routines for Accessing Waves** on page 253 and **Routines for Accessing Data Folders** on page 300.

For storing your own private data, you can use WM memory XOPSupport routines or standard C/C++ memory management routines. For data exchanged with Igor, you must use WM memory XOPSupport routines.

As explained under **Updating Old Code to Use WM Memory XOPSupport Routines** on page 199, 64-bit Macintosh XOPs created with XOP Toolkit 7.00 must be modified and recompiled to work with Igor Pro 8 and later on Macintosh.

### Handle Example

Before we get into details, let's start with a simple example using a Handle. This example sets the note property of a wave using the **SetWaveNote** callback routine.

```
int
DemoSetWaveNote()
{
    int err = 0;

    waveHndl waveH = FetchWave("wave0");
    if (waveH == NULL)
        return NOWAV;          // Wave does not exist

    char noteText[256];
    strcpy(noteText, "This is a test");
    int len = (int)strlen(noteText);

    Handle noteH = WMNewHandle(len);
    if (noteH == NULL)
        return NOMEM;

    err = PutCStringInHandle(noteH, noteText);
    if (err != 0) {
        WMDisposeHandle(noteH);
        return err;
    }

    err = SetWaveNote(waveH, noteH);    // Igor now owns the noteH Handle

    return err;
}
```

This routine uses WMNewHandle to create a Handle, PutCStringInHandle to set the contents of the Handle, and WMDisposeHandle to free the handle.

The next section explains why the XOP does not dispose noteH after passing it to SetWaveNote.

### Ownership of Handles

When a Handle is passed between Igor and an XOP, in some cases the caller retains ownership of the handle and is responsible for disposing it. In other cases, the callee takes ownership of the handle and is responsible for disposing it. In the preceding example, once the call to **SetWaveNote** is made, Igor owns the handle so the XOP does not dispose it. The XOP Toolkit documentation for each XOPSupport routine that uses a Handle states the ownership rules for that routine.

## Commonly-Used WM Memory XOPSupport Routines

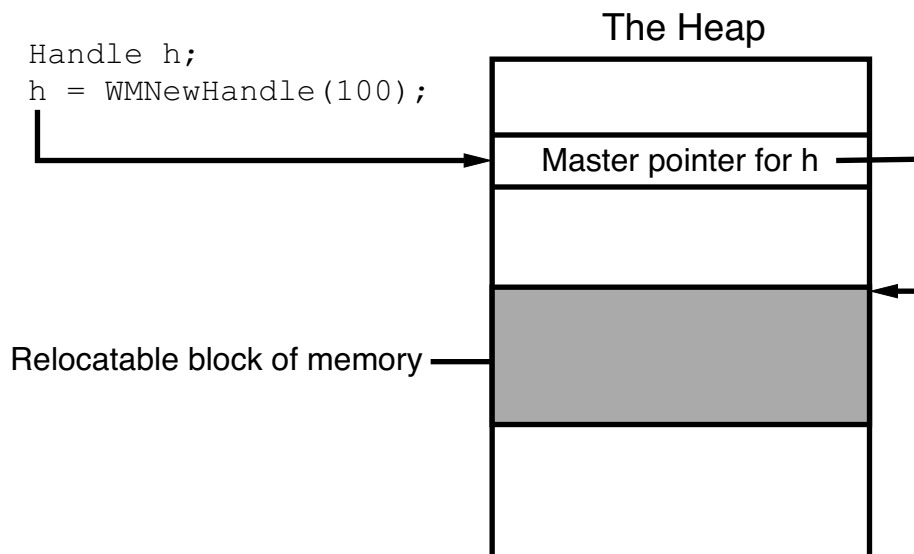
This table lists the WM memory XOPSupport routines that are commonly used in XOP programming. These routines, as well as some less commonly used routines, are described in detail in the section **WM Memory Routines** on page 410.

Routine	What It Does
Handle <b>WMNewHandle</b> (BCInt size);	Allocates a block of memory in the heap and returns a handle to it.
BCInt <b>WMGetHandleSize</b> (Handle h);	Returns the number of bytes in the block of memory referred to by the handle.
void <b>WMSetHandleSize</b> (Handle h, BCInt size);	Sets the number of bytes in the block of memory referred to by the handle.
void <b>WMDisposeHandle</b> (Handle h);	Frees the block of memory pointed to by the handle.

## Understanding Handles

The WMNewHandle function allocates a block of memory on the heap and returns a Handle which references that block. Unlike a pointer, which points directly to a block of memory, a Handle points indirectly to the allocated memory block.

Specifically, a Handle 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. This illustration shows the relationship of a Handle and the block of memory that it refers to:



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 WMNewHandle, then \*h is a pointer to the block of memory that h refers to.

In this example, we call WMNewHandle to allocate a two-byte block of memory. We set the first byte to 0 and the second byte to 1. We then dispose the Handle.

```
Handle h = WMNewHandle(2);
if (h == NULL)
    return NOMEM;
char* p = (char*)*h;    // DEREFERENCE
*p = 0;
```

```
*(p+1) = 1;
WMDisposeHandle(h);
```

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

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.

### Using Handles

In the 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”.

Writing to memory referenced by a dangling pointer is a disaster. It creates intermittent bugs that sometimes cause crashes well after the bad write. Finding the cause of such a bug is very difficult and time-consuming.

The problem of dangling pointers is greatly reduced because, unlike in times of old, blocks of memory are no longer relocated to satisfy memory allocation requests involving other blocks.

### Resizing Handles

The problem of dangling pointers is not completely gone. If you increase the size of a handle by calling `WMSetHandleSize`, 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, `WMSetHandleSize` 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. `WMSetHandleSize` ultimately calls the C `realloc` function or its 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:

```
int err = 0;

Handle h = WMNewHandle(2);
if (h == NULL)
    return NOMEM;
char* p = (char*)*h;          // DEREFERENCE
*p = 0;
*(p+1) = 1;

err = WMSetHandleSize(h,4);    // Increase size of handle
if (err == 0) {                // Make sure resize succeeded
    p = *h;                     // RE-DEREFERENCE
    *(p+2) = 2;
    *(p+3) = 3;
}
```



```
WMDisposeHandle(h);
return err;
```

For further discussion of this issue, see **Dangling Pointers** on page 228.

## Updating Old Code to Use WM Memory XOPSupport Routines

This section explains a compatibility issue that affects Macintosh 64-bit XOPs running with Igor Pro 8 (in development as of this writing in May of 2017) or later. The bottom line is:

- 64-bit Macintosh XOPs created with XOP Toolkit 7.00 must be modified and recompiled with XOP Toolkit 7.01 or later to run with Igor Pro 8
- Other XOPs (32-bit Macintosh XOPs and 32-bit and 64-bit Windows XOPs) do not need to be modified, but we recommend that you modify them when you next update such XOPs

Details of the required modifications are provided below. First, here is background information to help you understand the issue.

Historically on Macintosh, Igor and XOPs always used native Macintosh Handles and consequently used native Mac OS routines such as `NewHandle`, `GetHandleSize`, `SetHandleSize`, and `DisposeHandle`. On Windows, they always used WaveMetrics emulation routines with the same names, implemented in Igor and exported to XOPs through the `IGOR.lib` and `IGOR64.lib` files.

Native Macintosh Handles are limited by Mac OS to roughly 2 GB. This is unfortunately true even when running a 64-bit application. WaveMetrics emulation of Macintosh Handles never had this limitation.

Igor Pro 7 was the first version of Igor that ran in 64 bits on Macintosh and XOP Toolkit 7.00 was the first version of the XOP Toolkit that supported compilation of 64-bit Macintosh XOPs.

Working with the 64-bit version of MacIgor7 over time, we noticed that the Mac OS 2 GB limitation sometimes caused problems. To overcome this limit, starting with Igor8, the Macintosh 64-bit version of Igor also uses emulation of Handles. This table summarizes Igor's use of Handles:

	Igor Pro 7 and before	Igor Pro 8 and later
<b>Macintosh 32 Bit</b>	Native Macintosh	Native Macintosh
<b>Macintosh 64 Bit</b>	Native Macintosh	WaveMetrics Emulation
<b>Windows 32 Bit</b>	WaveMetrics Emulation	WaveMetrics Emulation
<b>Windows 64 Bit</b>	WaveMetrics Emulation	WaveMetrics Emulation

As the table shows, the only change is that, as of Igor Pro 8, Igor uses its own emulation of Handles instead of Mac OS native Handles when running on Macintosh in 64 bits.

For Handles passed between Igor and XOPs, such as for string parameters and string result, Igor and XOPs must use the same implementation of Handles. For example, if Igor allocates such a Handle to pass a parameter to an XOP, the XOP needs to call a function to determine the number of bytes associated with the Handle. The Mac OS native `GetHandleSize` function works only with Handles allocated by the Mac OS native `NewHandle` function. It does not work with Handles allocated by WaveMetrics' emulation of `NewHandle`. The converse is also true. So, when it comes to Handles, Igor and XOPs must use the same set of functions. Either both must use native Mac OS functions or both must use WaveMetrics emulation functions.

To make this possible, in XOP Toolkit 7.01, we introduced new routines named `WMNewHandle`, `WMGetHandleSize`, and so on. XOPs that use these WaveMetrics routines call the correct functions regardless of Igor version and operating system. See **WM Memory XOPSupport Routines** on page 195 for background information.

Because of the change to Handles in MacIgor64, when running in 64 bits on Macintosh, Igor8 and later report an error if you attempt to load an XOP created with XOP Toolkit 7.00. You must modify such XOPs

## Chapter 12 — Other Programming Topics

---

to use WaveMetrics memory XOPSupport routines instead of Mac OS native routines and recompile them using XOP Toolkit 7.01 or later. Existing Macintosh 32-bit XOPs and existing Windows 32-bit and 64-bit XOPs continue to work as before.

Windows-only XOPs can be compiled without changes in XOP Toolkit 7.01. However, for consistency we recommend that you update even Windows-only XOPs.

To update your XOP to use WM memory XOPSupport routines with XOP Toolkit 7.01 and later, follow these steps:

1. Replace all NewHandle calls with WMNewHandle.
2. Replace all GetHandleSize calls with WMGetHandleSize.
3. Replace all SetHandleSize calls with WMSetHandleSize.
4. Replace all HandToHand calls with WMHandToHand.
5. Replace all HandAndHand calls with WMHandAndHand.
6. Replace all DisposeHandle calls with WMDisposeHandle.
7. Replace all NewPtr calls with WMNewPtr.
8. Replace all GetPtrSize calls with WMGetPtrSize.
9. Replace all SetPtrSize calls with WMSetPtrSize.
10. Replace all PtrToHand calls with WMPtrToHand.
11. Replace all PtrAndHand calls with WMPtrAndHand.
12. Replace all DisposePtr calls with WMDisposePtr.
13. Remove all calls to MemError (details below).
14. Recompile your XOP.

Most XOPs call only a few of these routines so the number of changes required will typically be small.

Check your changes carefully. If you fail to replace, for example, NewHandle with WMNewHandle, you will get a link error when you build on Macintosh. The error occurs because of #defines in XOPMacSupport.h designed to make you aware if you forget to replace a Mac OS native call with the corresponding WM memory XOPSupport routine call.

The resulting XOP will continue to work correctly with versions of Igor6 or Igor7 that it previously worked with. The only difference is that it will now also work correctly with Igor8 in 64 bits on Macintosh.

There are some differences between the WM memory XOPSupport routines and the corresponding Macintosh routines:

1. WMSetHandleSize and WMSetPtrSize return an error code while the corresponding native Macintosh routines do not.
2. There is no WMMemError function corresponding to the native Macintosh MemError function. If you have a call to MemError, replace it like this:

Old	New
<pre>SetHandleSize(h, newSize); err = MemError();</pre>	<pre>err = WMSetHandleSize(h, newSize);</pre>

3. WMDisposeHandle and WMDisposePtr do nothing if passed a NULL parameter while the corresponding native Macintosh routines treat this as an error (nilHandleErr) which MemError returns.

It is unlikely that you call any native Mac OS routines that return Handles. However if you do, you must treat such native Mac OS Handles differently. You must use native Mac OS functions, not WaveMetrics memory XOPSupport routines, on native Mac OS Handles. In this case, you must #define ALLOW\_MAC\_OS\_NATIVE\_MEMORY\_ROUTINES as otherwise the #defines in XOPWinMacSupport.h will prevent you from using native Mac OS functions.

If you find it necessary to define `ALLOW_MAC_OS_NATIVE_MEMORY_ROUTINES`, be very careful. If you pass a Mac OS native Handle to Igor, or if you pass a WM handle to Mac OS, you are likely to corrupt memory, causing a crash that is very hard to track down.

## 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 MenuXOP1, NIGPIB2, SimpleFit, SimpleLoadWave, 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.

XOPs that add dialogs, such as GBLoadWaveX, require platform-specific files. See **Adding Dialogs** on page 207 for details.

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. See **Adding Windows** on page 155 for details.

## Text Encodings

A text encoding is a mapping from a set of numbers to a set of characters. For example, in the ASCII text encoding the number 0x41 (decimal 65) maps to the character Upper Case Letter A. The text encodings that are commonly used with Igor6 are MacRoman, Windows-1252 (also called "Western") and Shift JIS (Japanese). Igor Pro 7 and later use UTF-8. For further background information, execute this in Igor7 or later:

```
DisplayHelpTopic "Text Encodings"
```

In Igor Pro 6 and earlier, Igor used the "system text encoding" to represent text.

On Macintosh you set the system text encoding by setting the preferred language in the Languages&Region control panel. For most Macintosh Igor users the system text encoding is MacRoman or Shift JIS.

On Windows the system text encoding is called the "system locale" or "language for non-Unicode programs". You set it in the Administrative tab of the Region control panel. For most Windows Igor users the system text encoding is Windows-1252 or Shift JIS.

Starting with Igor Pro 7, Igor uses Unicode instead of the system text encoding. Specifically Igor7 uses UTF-8 — a Unicode encoding form that represents a given character using one, two, three or four bytes. Unicode allows you to access any character, rather than a limited set, and works the same on Macintosh, Windows, and any other platform.

Like most text encodings, UTF-8 uses the same one-byte codes as ASCII to represent the ASCII characters (represented by codes from 0x00 to 0x7F). Consequently, if you use ASCII characters only, you do not need to be concerned with text encoding issues. Non-ASCII characters, including accented characters, special symbols, Greek letters, and Asian characters, are represented using two, three or four bytes and are different in UTF-8 than in other text encodings.

Except as otherwise specified in the documentation for a given Igor message or XOPSupport routine, all string parameters are assumed to be encoded using system text encoding when running with Igor6 and as UTF-8 when running with Igor7 or later.

In most cases text that you pass to an XOPSupport routine originates in Igor and therefore uses the appropriate text encoding for the Igor version that you are running, so you don't need to worry about text encod-

ings. However if the text that you pass to an XOPSupport routine does not originate in Igor then you must make sure that it uses the right text encoding for the version of Igor that you are running.

One example of text that does originate in Igor is literal text in your program, such as a string that you might print in Igor's history area using the XOPNotice callback. This is discussed below under **Text Encodings in Source Files** on page 202.

For another example, if you are running with Igor7, and you read text from a file containing non-ASCII characters encoded as Windows-1252, and you store that text in an Igor text wave (using a callback such as **MDSetTextWavePointValue**), you must convert from Windows-1252 to UTF-8 and pass the UTF-8 text to the callback. Otherwise you will store text in the wave that is not valid UTF-8 text for the non-ASCII characters. This will lead to incorrectly rendered text later when the user displays the text and may cause errors when manipulating it. You can use the **ConvertTextEncoding** callback to do the conversion.

Since ASCII text (codes from 0x00 to 0x7F) is the same in nearly all text encodings that might be used on personal computers, text encoding is an issue only for non-ASCII text such as accented characters, special symbols, Greek letters, and Asian characters.

In the MDSetTextWavePointValue example, if you know that the file contains only ASCII text then you don't need to do any conversion because UTF-8 uses the same codes as ASCII for ASCII characters. However if the file might contain non-ASCII text, such as accented characters, if you fail to convert to UTF-8 you will store a byte sequence in the Igor text wave that is illegal in UTF-8. This will cause incorrect rendering or errors later.

Here are XOPSupport routines that relate to text encoding issues. They all require Igor Pro 7.00 or later:

Routine	Description
<b>ConvertTextEncoding</b>	Converts text from one text encoding to another.
<b>WaveTextEncoding</b>	Returns the text encoding code for an element of a wave. Most XOPs will not need to use this routine.

### Igor Pro 7 Text Encoding Codes

The routines listed above use Igor-specific codes to represent text encodings. These codes are defined by the WMTextEncodingCode enum which is defined in IgorXOP.h. The codes most commonly used with Igor are:

WMTextEncodingCode Enum	Description
kWMTextEncodingUTF8	UTF-8.
kWMTextEncodingMacRoman	Used for western text on Macintosh.
kWMTextEncodingWindows1252	Used for western text on Windows.
kWMTextEncodingJIS	Shift JIS, used for Japanese text.
kWMTextEncodingBinary	This is not a real text encoding. Rather it is used to mark the text data of a text wave that really contains binary data.

## Text Encodings in Source Files

The material in this section is relevant if your XOP uses non-ASCII text.

XOP Toolkit 7 projects are configured to use UTF-8 text encoding so that programmers can enter literal strings using any character in a cross-platform manner. This affects Igor6 and Igor7 differently:

1. Literal text in UTF-8 source files is compatible with Igor7 or later.
2. Literal text containing non-ASCII characters in UTF-8 source files is not compatible with Igor6.

If you enter a literal string containing non-ASCII text in a source file configured as UTF-8, it will be correct for use with Igor7. For example:

```
XOPNotice("This is a bullet character: •" CR_STR);
```

This will produce the correct bullet character with Igor7 or later on any platforms.

It will produce an incorrect character with Igor6 on all platforms.

If producing incorrect characters in Igor6 is not acceptable, you need to include code to support multiple text encodings. For example, the XOPNotice statement above can be rewritten like this:

```
char message[256];
*message = 0;
if (igorVersion < 700) {
    #ifdef MACIGOR
        // 0xA5 is bullet in MacRoman
        strcpy(message, "This is a bullet character: " "\xA5" CR_STR);
    #endif
    #ifdef WINIGOR
        // 0x95 is bullet in Windows-1252
        strcpy(message, "This is a bullet character: " "\x95" CR_STR);
    #endif
}
else {
    // \xE2\x80\xA2 is bullet in UTF-8
    strcpy(message, "This is a bullet character: " "\xE2\x80\xA2" CR_STR);
}
XOPNotice(message);
```

This will work correctly for western Macintosh and Windows users in Igor6 and Igor7.

(If Igor6 is running with a Japanese font for the history area, this code will produce incorrect results because it does not take Shift JIS into account. There is no general way to handle this because you have no way to know what font the history area is using.)

So much code for something seemingly so simple is clearly unappealing. Here are some other strategies that you can consider, in order of increasing difficulty:

1. Eliminate non-ASCII characters.

This is by far the simplest solution. It side-steps all text encoding issues.

2. Live with incorrect text in Igor6.

With Igor6 you will see incorrect characters, but Igor6 will eventually be ancient history.

3. Freeze your Igor6 code and continue development for Igor7 only in a new branch.

Use system text encoding (MacRoman on Macintosh and Windows-1252 on Windows) in the Igor6 branch of your project and UTF-8 in the Igor7 branch.

As relates to text encoding, this manual and the XOP Toolkit 7 sample projects are optimized for use with Igor7 and later.

## **Text Encodings in Xcode**

In Xcode you can use the File Inspector text encoding setting to set the text encoding for a given source file to MacRoman, UTF-8 or any other text encoding. This per-file setting is stored in the project file. Consequently, it sticks if you share the project file with another programmer.

Xcode does not automatically set the text encoding of a file based on its contents, even if the file contains UTF-8 text, and even if the file starts with a UTF-8 byte order mark. It always treats the file using the encoding stored for the file in the project, as set by the Text Encoding popup menu in the File Inspector pane.

## Chapter 12 — Other Programming Topics

---

The XOP Toolkit 7 Xcode projects are configured to use UTF-8 for source files except for resource files (.r files and XOPResources.h). Resource files are configured as MacRoman because the Rez resource compiler used by Xcode does not support UTF-8.

Line endings are set in the Xcode projects to use CRLF for compatibility with Windows editors.

Unlike Visual Studio, Xcode does not provide a way to add a UTF-8 byte order mark to a source file. As explained below, in Visual Studio, files must include a UTF-8 byte order mark to guarantee that Visual Studio treats them as UTF-8. If you share files between Xcode and Visual Studio, you can add byte order marks as described in the next section or you can use the “Text Encoding Utilities.pxp” experiment, shipped in IgorXOPs7/Extra. This has already been done for the XOP Toolkit 7 sample projects.

The XOPResources.h file has no byte order mark because Apple’s Rez compiler does not support it.

### Text Encodings in Visual C++

In Visual Studio 2015 and 2017, you can set the text encoding for a given file by opening the file, choosing File→Save As, clicking the triangle next to the Save button, and choosing Save With Encoding. This displays the Advanced Save Options dialog. Once you choose the desired text encoding, you can proceed to finish the Save As operation.

The text encoding setting, as set in the Advanced Save Options dialog, is not saved in the project but merely remembered while the file is open in the editor pane. If you close and reopen the file, or close and reopen the project, the text encoding setting for the file reverts to the default system text encoding, usually Windows-1252.

If you set the file text encoding to “UTF-8 with signature”, Visual Studio adds a UTF-8 byte order mark to the file. (“signature” is Microsoft terminology for “byte order mark”.) If you then save the file, close it and reopen it, Visual Studio recognizes the UTF-8 byte order mark and treats the file as UTF-8. You can also add byte order marks using the “Text Encoding Utilities.pxp” experiment, shipped in IgorXOPs7/Extra.

We want to use UTF-8 when running with Igor7 or later, so all of the .c, .cpp, and .h source files included with the XOP Toolkit have been set to include the UTF-8 byte order mark. Resource files (.rc and resource.h) are configured as Windows-1252 because the Windows resource compiler does not support UTF-8.

In addition to making the Visual Studio editor use UTF-8, we need to make the Visual C++ compiler use it. To do that, we include the /utf-8 flag in the Command Line section of the C/C++ section of the property pages for all configurations of the project. The /utf-8 flag is discussed at <https://msdn.microsoft.com/en-us/library/mt708821.aspx>. As explained at [support.microsoft.com/en-us/kb/980263](https://support.microsoft.com/en-us/kb/980263), this setting requires Visual Studio 2015 Update 2 or later. The /utf-8 flag affects the compiler only - it does not force the editor to treat the file as UTF-8.

## File I/O

The XOP Toolkit provides the following platform-independent routines for creating, deleting, reading, writing, and otherwise dealing with files:

<b>FullPathPointsToFile</b>	<b>FullPathPointsToFolder</b>	
<b>XOPCreateFile</b>	<b>XOPDeleteFile</b>	
<b>XOPOpenFile</b>	<b>XOPCloseFile</b>	
<b>XOPReadFile</b>	<b>XOPReadFile2</b>	<b>XOPReadLine</b>
<b>XOPWriteFile</b>		
<b>XOPGetFilePosition</b>	<b>XOPGetFilePosition2</b>	
<b>XOPSetFilePosition</b>	<b>XOPSetFilePosition2</b>	

---

**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).

When running with Igor6, file name and path parameters that you pass to these routines must be encoded using system text encoding. When running with Igor7, they must be encoded as UTF-8. If the file name or path is ASCII then this requirement is satisfied. Otherwise, if you receive the file name or path from Igor as a parameter to an external function or operation then this requirement is satisfied.

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 GBLoadWaveX illustrate the use of these routines.

If your XOP requires Igor7 or later, you can use the newer **XOPOpenFileDialog2** and **XOPSaveFileDialog2** routines which are platform-independent rather than the older routines which have different parameters on Macintosh and Windows.

## 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 MacOS, POSIX paths with forward slashes are often used. However, Igor and the XOP Toolkit, for historical reasons, use 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 MacOS 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.

## Long Names and Paths in XOPs

In Igor Pro 8.00 and later, Igor supports long object names and long file system paths.

Object names, as well as dimension labels, which were previously limited to 31 bytes, can now be up to 255 bytes.

File system paths, which were previously limited to 511 bytes on Macintosh and 259 bytes on Windows, can now be up to 2000 bytes, subject to operating system limits.

Igor commands, which were previously limited to 1000 bytes, can now be up to 2500 bytes.

Igor Pro 8 still supports existing XOPs, created with XOP Toolkit 7 and before, with the original name and path length limitations so long as the user does not pass long names or paths to the XOP. If the user attempts to pass a long name or path to an existing XOP, or if the XOP tries to access an object with a long name, Igor generates an error.

If you are targeting Igor Pro 8.00 or later, and you are creating a new XOP or updating an XOP already updated using XOP Toolkit 7.01 or later, you should use XOP Toolkit 8. See **Choosing Which XOP Toolkit to Use** on page 5 for details.

Once you are ready to update your XOP using XOP Toolkit 8, see "Long Names and Paths in XOPs" in Chapter 12 of the XOP Toolkit 8 manual for further details.

## Alerts and Message Boxes

The following routines provide a cross-platform way to display simple alert dialogs.

Routine	What It Does
<b>XOPOKAlert</b>	Displays a message with an OK button.
<b>XOPOKCancelAlert</b>	Displays a message with OK and Cancel buttons.
<b>XOPYesNoAlert</b>	Displays a message with Yes and No buttons.
<b>XOPYesNoCancelAlert</b>	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.
XOPOpenFileDialog2	Displays a standard Open File dialog. Requires Igor7 or later.
XOPSaveFileDialog	Displays a standard Save File dialog.
XOPSaveFileDialog2	Displays a standard Save File dialog. Requires Igor7 or later.

## Adding Dialogs

When the user selects your XOP's menu item you may want to put up a dialog. The GBLoadWaveX sample XOP does this and provides a good starting point for your XOP dialog.

Prior to XOP Toolkit 7, the XOPSupport library included cross-platform dialog support routines. Because the programming techniques used for these routines are obsolete on Macintosh, they were removed from the XOP Toolkit 7 XOPSupport library on both Macintosh and Windows. Now you must implement dialogs separately for Macintosh and Windows.

The XOP Toolkit includes an Extras folder containing files that are not part of the XOPSupport library but which may be of use. The GBLoadWaveX and VDT2 XOPs, which create dialogs, use the DialogUtilities files from the Extras folder to support dialog creation and management. On Windows, the DialogUtilities-Win.c file provides most of the routines that were removed from the XOPSupport library itself. These routines enable you to create a Windows dialog in XOP Toolkit 7 using nearly the same code as used in XOP Toolkit 6. On Macintosh, the DialogUtilitiesMac.c file provides a small subset. This is because dialog creation on Macintosh changed radically with the advent of Cocoa so the old routines are no longer of use.

On Macintosh, you implement a dialog using Apple's Cocoa framework. The GBLoadWaveX sample XOP provides a good starting point. If you are unfamiliar with Cocoa, you will need to familiarize yourself with it.

On Windows, you can write your dialog using standard Windows programming techniques. The GBLoadWaveX sample XOP provides a good starting point.

If you are updating old XOP dialog code, see **XOP Dialogs in XOP Toolkit 7**.

## 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) {
    0x01, 0x00, release, 0x00, 0,    // Version of XOP
    "1.00",                          // Version bytes and country code
    "1.00, © 2012 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 6.20 or later)"
};
```

### Windows Version Resources

You create a version resource by editing the VERSIONINFO 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 7 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 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

You can create external functions and external operations that take structure 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 and you must use two-byte packing for the structure. Otherwise a crash is likely to occur.

For examples, see **External Operation Structure Parameter Example** on page 90 and **External Function Structure Parameter Example** on page 112.

### 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 <b>Strings In Structures</b> on page 210.
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	
int64	SInt64	Requires Igor7 or later
uint64	UInt64	Requires Igor7 or later

Igor Field Type	C Field Type	Notes
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 211. 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 212 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 **WMGetHandleSize** 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 latter 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 233 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 131.

### 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 143.

## 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 = WMNewHandle(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:
    WMDisposeHandle(ourStrH);    // OK if NULL
    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.

**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 Preemptive 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 preemptive 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 preemptive thread, you must 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 preemptive 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. A procedure that is defined outside of any independent module exists by default in the built-in **ProcGlobal** module. See the Igor Pro manual for details.

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 or in **ProcGlobal**. If you call **GetFunctionInfo** with a fully-qualified name, such as "ProcGlobal#MyFunction" or "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**. If you want to call a function in **ProcGlobal**, it is a good idea to specify it in the function name, as illustrated in the example below.

**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 coefsWaveH)
{
    FunctionInfo fi;
    OurParams parameters;
    int badParameterNumber;
    int requiredParameterTypes[2];
```

```
double result;
int i;
double values[5];
int err;

// Make sure the function exists and get information about it.
if (err = GetFunctionInfo("ProcGlobal#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 = coefsWaveH;
for(i=0; i<5; i+=1) {
    parameters.x = i;
    if (err = CallFunction(&fi, (void*)&parameters, &result))
        return err;
    values[i] = result;
}

return 0;
}
```

## 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 **CheckAbort**.

SpinProcess is so-named because originally it spun the beachball cursor during a lengthy calculation on Macintosh. Now the operating system spins the beachball if the program stops responding to events. However, in Igor7 or later, SpinProcess still spins the beachball in the bottom/right 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
- 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.

**NOTE:** Calling SpinProcess, or any other callback, may cause recursion. This can cause your XOP to crash if you do not handle it properly. See **Handling Recursion** on page 67 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, use SpinProcess instead of CheckAbort.

## 64-bit Integer Issues

Igor Pro 7 added the ability to create signed and unsigned 64-bit integer waves. The XOP Toolkit supports such waves, but there are many issues.

*For almost all applications, you should use double waves rather than 64-bit integer waves.*

Doubles (64-bit floating point) can represent integers precisely if the integer fits in 53 bits. Thus, the largest integer that is precisely represented by a double is  $2^{53}-1$  which equals 9,007,199,254,740,991 which is roughly 9 quadrillion.

Doubles represent integers larger than 9 quadrillion approximately. This means that any time Igor, your XOP, or Igor procedure code stores a very large integer in a double, precision is lost. This is usually not a problem because it is very rare for programs to need to represent integers larger than 9 quadrillion.

Internally, Igor uses doubles for most calculations. Consequently, if you use 64-bit integer waves in calculations, it takes time to do this conversion and you lose precision to boot. Furthermore, the conversion of very larger integers from 64-bit integer to double may cause internal overflow errors giving incorrect results.

Here is an example showing an overflow error:

```

Function Demo()
    Make/O/D/N=2 FP64
    Make/O/L/N=2 SInt64
    Edit FP64, SInt64
    ModifyTable format(FP64)=1, width(FP64)=150
    ModifyTable format(SInt64)=1, width(SInt64)=150

    FP64[0] = 9007199254740991           // Represented precisely
    FP64[1] = 9223372036854775807       // Represented approximately
    SInt64[0] = FP64[0]                 // Correct
    SInt64[1] = FP64[1]                 // Incorrect due to overflow
End

```

## Chapter 12 — Other Programming Topics

---

This produces the following table:

	<b>FP64</b>	<b>SInt64</b>
0	9007199254740991	9007199254740991
1	9223372036854775808	-9223372036854775808

The statement:

```
FP64 [1] = 9223372036854775807
```

gave us a value of 9223372036854775808 for FP64[1] because 9223372036854775807 can not be precisely represented in a double.

The value of SInt64[1] is wrong. This statement

```
SInt64 [1] = FP64 [1]
```

has to convert from double-precision floating point to signed 64-bit integer. In this case, the value represented by FP64[1] is too big to fit in a signed 64-bit integer. Overflow occurs and we get an incorrect result.

In principal it is possible for the conversion from double to signed 64-bit integer to include clipping so that we would get 9223372036854775807 from the statement. However, clipping is time consuming, and to find and fix every place in Igor where clipping is needed is practically impossible and would likely introduce bugs.

Tables in Igor7 use doubles internally and thus do not properly handle the entry or display of very large integers. In Igor8, tables properly handle very large 64-bit integers.

The point of this section is that the use of 64-bit integers comes with pitfalls. This is why, for nearly all applications, you should use double rather than 64-bit integers.

Using 64-bit integer waves is appropriate if you are dealing with a data source that produces 64-bit integer data and if there is a chance that such data may exceed 9 quadrillion in magnitude. In this case, you can use 64-bit integer waves, but you should be aware of the issues raised in this section.

To retrieve or store 64-bit integer data from 64-bit integer waves, you must use **The Direct Access Method** (see page 126) or one of these routines which require Igor Pro 7.03 or later:

<b>MDGetNumericWavePointValueSInt64</b>	<b>MDSetNumericWavePointValueSInt64</b>
<b>MDGetNumericWavePointValueUInt64</b>	<b>MDSetNumericWavePointValueUInt64</b>

## 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 in C++ source files must 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)
```

### Using C++ Exceptions

If your C++ code uses C++ exceptions, all exceptions thrown from within your XOP must be caught inside your XOP. This includes exceptions thrown by C++ library routines that you call. Otherwise the exception will cause Igor to crash.

## Windows DLL Dependencies

In Igor Pro 6.20 or later, Igor loads Windows XOPs using the Windows OS LoadLibraryEx routine with the LOAD\_WITH\_ALTERED\_SEARCH\_PATH flag. This means that the OS will look for DLLs required by the XOP in the folder containing the XOP before looking elsewhere. You can ship a DLL that your XOP depends on in the same folder as the DLL and no further installation is required of the end user. This technique is explained at:

[http://msdn.microsoft.com/en-us/library/ms682586\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms682586(VS.85).aspx)

With earlier versions of Igor, the DLL has to be in placed in the folder containing Igor.exe or in a folder listed in the path environment variable.



## Providing Help

### Overview

If your XOP will be used by other people, you should provide help. This chapter explains how to do that.

Previous XOP Toolkits supported Macintosh balloon help and Windows menu help. These are now obsolete. If you are updating an old XOP, see **Help Resources in XOP Toolkit 7** on page 423 for details.

### 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 193.

### 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.

## Chapter 13 — Providing Help

---

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 "GBLoadWaveX.xop", Igor looks for "GBLoadWaveX 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 "GBLoadWaveX Release" and another XOP named "GBLoadWaveX Debug", and you want both XOPs to use a single help file named "GBLoadWaveX 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 GBLoadWaveX.

```
// Macintosh, in GBLoadWaveX.r.
resource 'STR#' (1101) {           // Misc strings that Igor looks for.
    {
        "-1",
        "---",
        "GBLoadWaveX Help",       // Name of XOP's help file.
    }
};

// Windows, GBLoadWaveXWinCustom.rc.
1101 STR#                          // Misc strings that Igor looks for.
BEGIN
    "-1\0",
    "---\0",
    "GBLoadWaveX 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.

As explained under **Command Help for 64-bit XOPs** on page 193, when running a 64-bit XOP, such as GBLoadWaveX64.xop, Igor will look for a help file named using 32-bit conventions, such as "GBLoadWave Help.ihf".

## Programmatically Displaying a Help Topic

You may want to display a help topic from your help file when the user clicks a Help button in a dialog or at some other time. You can do this by calling **XOPDisplayHelpTopic** XOPSupport routine.

# 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 proofread your code.

#### Excessive Use of Global Variables

Global variables are bad because any routine in your program can change them and any routine can depend on them. This can lead to complex and unexpected dependencies which in turn leads to 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 your 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*)WMNewPtr(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 WMNewPtr. 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 = WMNewHandle(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 of the aHandle variable. If the first byte of aHandle was already 0, this will be asymptomatic. If it was not 0, it will likely cause a crash when aHandle is used or disposed. The crash will be intermittent, making it difficult to fix.

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*)WMNewPtr(sizeof(int));
    if (ip1 != NULL) {
        F1(ip1);
        WMDisposePtr((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 `WMNewPtr`.

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 = WMNewHandle(size);
        <Do something with the handle>;
    }

    WMDisposeHandle(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. If it happens to be 0 (NULL), then it will be asymptomatic. Otherwise, it is likely to crash.

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 = WMNewHandle(size);
        <Do something with the handle>;
    }
}

```

```
    if (h != NULL)                // Test flag
        WMDisposeHandle(h);
    return err;
}
```

In this particular case, the test for `h` being `NULL` is not needed as `WMDisposeHandle` does nothing if its parameter is `NULL`.

### 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 = WMGetHandleSize(gH); // Uses global before it is initialized.
    memset(**gH, 0, size);
}

void
InitializePart2(void)
{
    gH = WMNewHandle(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 MacOS, 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 directly, i.e., using `WMDisposeHandle`.

String handles that Igor passes to you sometimes remain owned by Igor and sometimes become owned by you. For example, string handles passed as parameters to your external function are owned by you and you must dispose them. But string handles passed as parameters to your external operations remain owned by Igor and you must not dispose them. Consult the XOP Toolkit documentation for the feature or XOPSupport routine you are using. It tells you when a handle belongs to Igor and when it is yours to dispose.

A similar situation arises with libraries that you might use. Whenever you receive a pointer or handle from a library, you must be certain about who is responsible for eventually deleting it. 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 = WMNewHandle(10000);
    memset(*h, 0, 10000);          // Possible bug
}
```

Under sufficiently low memory, the `WMNewHandle` 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 = WMNewHandle(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()
{
    char* p = (char*)malloc(100);
    if (p == NULL)
        return NOMEM;
    memset(p, 0, 100);
    . . .
    free(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 useful technique is to use the pointer itself as a flag to determine if it is valid or not:

```
int
Good()
{
    char* p = (char*)malloc(100);
    if (p == NULL)
        return NOMEM;
    memset(p, 0, 100);
    . . .
    free(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 = (char*)malloc(100);
        if (p == NULL)
            return NOMEM;
    }
    memset(p, 1, 100);
}
```

```

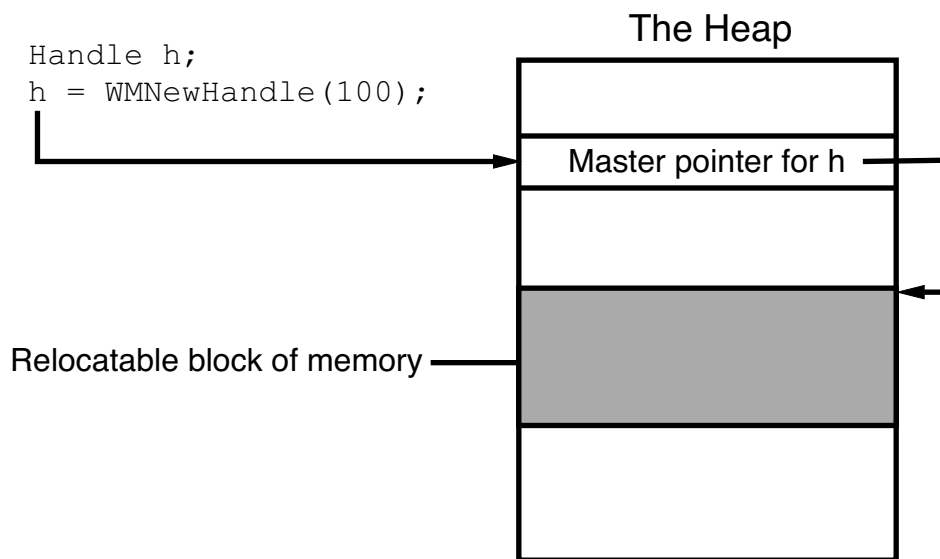
    . . .
    free(p);
    p = NULL;                // Set to NULL to signify no longer valid
    return 0;
}

```

The last `p=NULL` statement is 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

As explained under **Understanding Handles** on page 197, a 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.

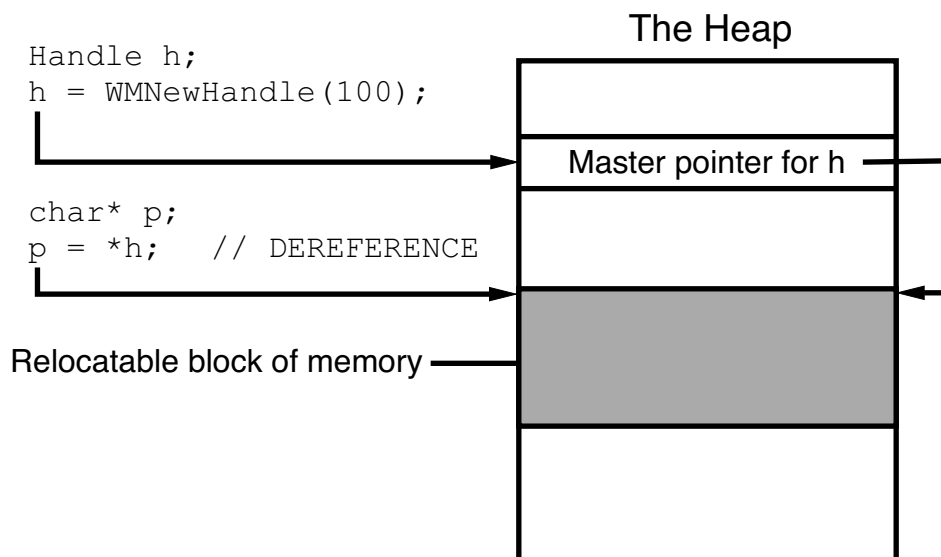


Igor uses Handles mostly for passing variable-length data, typically strings, to an XOP, and for receiving variable-length data from it. You use routines described under **WM Memory XOPSupport Routines** on page 195 to deal with Handles.

As the following sections explain, you must be careful when using a Handle to avoid using a dangling pointer to its 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 for a different object. 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 `WMSetHandleSize` 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;
int err = 0;

h1= WMNewHandle(100);
p = *h;                                // Dereference h to point to data.
memset(p, 0, 100);

. . .
err = WMSetHandleSize(h, 200);          // Increase size of heap block
if (err != 0)
    return err;
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 `WMSetHandleSize` may cause the memory manager to relocate the block of memory that `h` refers to.

### Dereferencing Wave Handles

A wave handle is like a `Handle` but you must always use wave-specific `XOPSupport` routines, described under **Routines for Accessing Waves** on page 253, to access it. Never use generic WM memory routines to access waves.

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 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 a 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 or MDAccessNumericWaveData function to dereference them.

### Recommendations for Using Handles

Now that we know 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 **WMGetHandleSize**.

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;
int err = 0;

h = <A routine that returns a string in a handle>;
if (h != NULL) {
    len = WMGetHandleSize(h);
    err = WMSetHandleSize(h, len+1); // Make room for null terminator.
    if (err != 0)
        return err;
    *(*h+len) = 0; // Null terminate the text.
    p = *h;
    <Use p as a C string>;
    WMSetHandleSize(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 handles and data folder handles, Igor always owns these. You must never dispose of them or otherwise directly modify them using WM memory XOPSupport routines. Instead, always use the specific routines for a given type of object. These routines are listed under **Routines for Accessing Waves** on page 121 and **Routines for Accessing Data Folders** on page 300.

### 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 74 and **Choose a Distinctive Function Name** on page 102 for details.

### Watch Out for Recursion

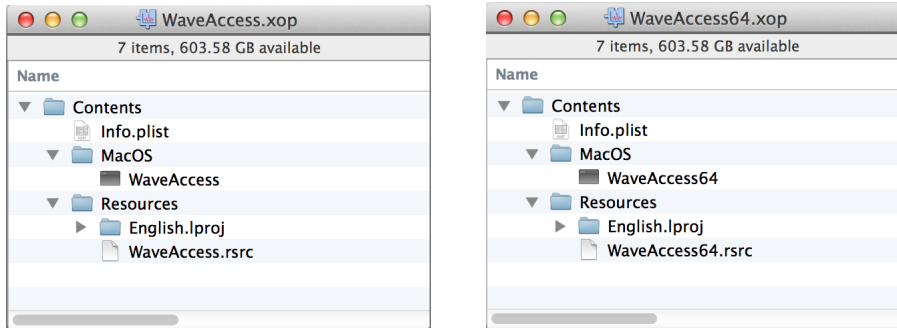
If your XOP adds a window to Igor, the DoUpdate, XOPCommand, XOPSilentCommand, and SpinProcess XOPSupport routines can cause Igor to call your XOP while your XOP is already running. See **Handling Recursion** on page 67 for details.

### Structure Alignment

Make sure that all structures that might be shared with Igor are 2-byte-aligned. See **Structure Alignment** on page 208 for details.

### Check Your Info.plist File

As explained under **XOP Projects in Xcode** on page 36, 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). 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”.

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 Xcode. One of the keys in the XML file is named CFBundleExecutable. In Xcode the CFBundleExecutable key is usually shown using its friendly name - “Executable file”.

When the XOP is compiled, Xcode copies the Info.plist file from the project to Contents/Info.plist in the package. Thus, you should always edit the Info.plist file in the project, not the one in the package.

For a 64-bit XOP, Contents/Info.plist in the package folder is copied from the project's Info64.plist file. Thus, you must set the CFBundleExecutable key in the project's Info64.plist file (e.g., **WaveAccess64.xop**) to the appropriate 64-bit name (e.g., **WaveAccess64**).



## XOPSupport Routines

### Table of Contents

About XOPSupport Routines.....	238
Routines for Communicating with Igor.....	239
Operation Handler Routines .....	242
Routines for Accessing Waves .....	253
Routines for Accessing Variables.....	291
Routines for Accessing Data Folders.....	300
Routines for XOPs with Menu Items.....	316
Routines for XOPs that Have Dialogs .....	327
Routines for XOPs that Access Files.....	328
Routines for File-Loader XOPs .....	345
Routines for XOPs with Windows.....	349
Routines for XOPs with Text Windows .....	350
Routines for Dealing with Resources .....	351
Routines for XOPs That Use FIFOs .....	353
Numeric Conversion Routines.....	354
Routines for Dealing With Object Names .....	358
Color Table Routines .....	364
Command Window .....	366
Routines for Dealing With Igor Procedures .....	370
Multithreading Support Routines.....	379
Windows-Specific Routines .....	382
Miscellaneous Routines .....	384
Programming Utilities.....	399
WM Memory Routines .....	410

### 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. See **Igor/XOP Compatibility Issues** on page 69 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 170 for further information.

See **The IORecHandle** on page 66 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 61 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 67 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 66.

*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 67 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 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 178 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 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 178 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 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 178 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 85 for further information.

*Example*

See VDTReadBinary.cpp.

*Thread Safety*

VarNameToDataType is Igor-thread-safe with 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 178 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 85 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 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 178 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 85 for further information.

### *Example*

See VDTReadBinary.cpp.

### *Thread Safety*

StoreNumericDataUsingVarName is Igor-thread-safe with 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 178 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 85 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 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 178 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 85 for further information.

### Example

See `VDTReadBinary.cpp`.

### Thread Safety

`StoreStringDataUsingVarName` is Igor-thread-safe with 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 178 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 86) 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 87 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 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 178 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 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 178 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, <b>GetOperationDestWave</b> returns an error.
kOpDestWaveChangeExistingWave	If the destination wave already exists its dimensions and data type are changed based on the <code>dimensionSizes</code> and <code>dataType</code> parameters but other properties (units, dimension scaling, dimension labels, wave note) are left unchanged.

Flag	Description
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 135 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 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 178 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 current and 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_I64	64 bit signed integer - requires Igor Pro 7.00 or later
NT_I8   NT_UNSIGNED	8 bit unsigned integer
NT_I16   NT_UNSIGNED	16 bit unsigned integer
NT_I32   NT_UNSIGNED	32 bit unsigned integer
NT_I64   NT_UNSIGNED	64 bit unsigned integer - requires Igor Pro 7.00 or later
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 the discussion of dangling pointers under **The Direct Access Method** on page 126.

### Thread Safety

MakeWave is Igor-thread-safe with Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 the discussion of dangling pointers under **The Direct Access Method** on page 126.



*Thread Safety*

ChangeWave is Igor-thread-safe with Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 131.

*Thread Safety*

KillWave is Igor-thread-safe with Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 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 178 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 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 178 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_I64	64 bit signed integer - Igor Pro 7.00 or later
NT_I8   NT_UNSIGNED	8 bit unsigned integer
NT_I16   NT_UNSIGNED	16 bit unsigned integer
NT_I32   NT_UNSIGNED	32 bit unsigned integer
NT_I64   NT_UNSIGNED	64 bit unsigned integer - Igor Pro 7.00 or later
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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 for details.

## WavePoints

```
Count Int
```

```
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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 for details.

## WaveNoteCopy

```
Handle
WaveNoteCopy(waveHandle)
waveHndl waveHandle;      // Handle to wave's data structure
```

Returns a handle containing the wave's note text or NULL if the wave has no wave note.

The returned handle is a copy of Igor's internal wave note handle. You own the returned handle and, if it is not NULL, you must dispose it using **WMDisposeHandle**.

Note that the text in the handle is *not* null terminated. Use **WMGetHandleSize** 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 233.

If you want to change the wave's note, use the **SetWaveNote** callback.

**WaveNoteCopy** is a replacement for the **WaveNote** routine from XOP Toolkit 6 and before. If you used **WaveNote**, change the call to **WaveNoteCopy** and, if the returned handle is not NULL, dispose the handle when you are finished using it.

Added in Igor Pro 7.00 but works with any version.

### *Thread Safety*

**WaveNoteCopy** is Igor-thread-safe with Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 **WMNewHandle** or **WMHandToHand**, 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 for details.

## WaveName

```
void
WaveName(waveHandle, name)
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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 for details.

## WaveTextEncoding

```
int
WaveTextEncoding(waveH, element, getEffectiveTextEncoding, tecPtr)
waveHndl waveH;
int element;
int getEffectiveTextEncoding;
int* tecPtr;
```

Returns via tecPtr a WMTextEncodingCode that identifies the text encoding of the stored text associated with the part of a wave identified by element.

WMTextEncodingCode is an enum defined in IgorXOP.h.

element is one of the following:

Value	Wave Element
1	Wave name
2	Wave units
4	Wave note
8	Wave dimension labels
16	Text wave contents

If `getEffectiveTextEncoding` is 1, `WaveTextEncoding` returns the effective wave text encoding. Otherwise it returns the raw wave text encoding. See the help for the built-in Igor7 `WaveTextEncoding` function for an explanation of this distinction.

The values returned via `tecPtr` are described in the documentation for the built-in Igor `WaveTextEncoding` function in Igor Pro 7. The most common values are:

WMTextEncodingCode	Description
kWMTextEncodingUTF8	UTF-8
kWMTextEncodingMacRoman	MacRoman (Macintosh western European)
kWMTextEncodingWindows1252	Windows-1252 (Windows western European)
kWMTextEncodingJIS	Shift-JIS (Japanese)
kWMTextEncodingBinary	

`kWMTextEncodingBinary` is not a real text encoding code but rather indicates that the wave contains binary data rather than text data.

Because nearly all XOPSupport routines, when running with Igor7, require text parameters to be UTF-8 and return output text as UTF-8, most XOPs will not need to use this routine.

See the **Text Encodings** on page 201 and the documentation for the built-in Igor7 `WaveTextEncoding` function for further information.

Added for Igor Pro 7.00. If you call this with an earlier version of Igor, it will return `IGOR_OBSOLETE` and do nothing.

### *Thread Safety*

`WaveTextEncoding` is Igor-thread-safe with XOP Toolkit 7 and Igor Pro 7.00 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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. To avoid dangling pointer bugs, see the discussion of dangling pointers under **The Direct Access Method** on page 126.

See Chapter 7 for detailed instructions on accessing wave data.

### *Thread Safety*

`WaveData` is Igor-thread-safe with Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 for details.



## WaveMemorySize

```
void*
WaveMemorySize(waveH, which)
waveHndl waveH;           // Wave handle
int which;                 // Code for what information you want
```

Most XOPs will not need to use this routine.

In Igor Pro 6 and before, wave handles were regular handles. Although there was little reason to do so, you could call WMGetHandleSize 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 WMGetHandleSize 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 84 for an example.

Returns 0 if OK or a non-zero error code.

### *Thread Safety*

CalcWaveRange is Igor-thread-safe with Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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
NT_I8	8 bit signed integer
NT_I16	16 bit signed integer
NT_I32	32 bit signed integer
NT_I64	64 bit signed integer - requires Igor Pro 7.00 or later
NT_I8   NT_UNSIGNED	8 bit unsigned integer
NT_I16   NT_UNSIGNED	16 bit unsigned integer
NT_I32   NT_UNSIGNED	32 bit unsigned integer
NT_I64   NT_UNSIGNED	64 bit unsigned integer - requires Igor Pro 7.00 or later
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.

Igor supports a maximum of four dimensions. Therefore, dimSizes[n] must be zero, where n is less than or equal to four. MAX\_DIMENSIONS is larger than 4 to allow XOPs to continue to work in the event that a future version of Igor supports more than four dimensions.

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 135 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 the discussion of dangling pointers under **The Direct Access Method** on page 126.

### 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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.

Igor supports a maximum of four dimensions. Therefore, dimSizes[n] must be zero, where n is less than or equal to four. MAX\_DIMENSIONS is larger than 4 to allow XOPs to continue to work in the event that a future version of Igor supports more than four dimensions.

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 the discussion of dangling pointers under **The Direct Access Method** on page 126.

#### 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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.

See **MDChangeWave** for further discussion.

**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 the discussion of dangling pointers under **The Direct Access Method** on page 126.

### Thread Safety

MDChangeWave2 is Igor-thread-safe with Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 129 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 129 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 129 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 129 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 must allocate dimLabel as follows:

```
char dimLabel[MAX_DIM_LABEL_BYTES+1];
```

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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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.
const char* dimLabel;     // You pass value here
```

Sets the label for the specified element of the specified dimension.

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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 **WMDisposeHandle** 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\_BYTES+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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 **WMDisposeHandle** 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 125 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 125 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.

Numeric wave data is stored contiguously in the wave handle in one of the supported data types (NT\_I8, NT\_I16, NT\_I32, NT\_I64, NT\_FP32, NT\_FP64). These types will be ORed with NT\_CMPLX if the wave is complex and ORed with NT\_UNSIGNED if the wave is unsigned integer. 64-bit integer waves (NT\_I64) were added in Igor Pro 7.00.

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.

Although they are not truly numeric waves, MDAccessNumericWaveData also allows you to access wave reference waves (WAVE\_TYPE) and data folder reference waves (DATAFOLDER\_TYPE) which store waveHndls and DataFolderHandles respectively.

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:

dimSizes[0]	Number of rows in a column
dimSizes[1]	Number of columns in a layer
dimSizes[2]	Number of layers in a chunk
dimSizes[3]	Number of chunks in the wave

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 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;
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;

double* dp = (double*)((char*)(*waveH) + dataOffset);
float* fp = (float*)dp;
SInt64* SInt64p = (SInt64*)dp;
SInt32* SInt32p = (SInt32*)dp;
short* sp = (short*)dp;
char* cp = (char*)dp;
UInt64* UInt64p = (UInt64*)dp;
UInt32* UInt32p = (UInt32*)dp;
unsigned short* usp = (unsigned short*)dp;
unsigned char* 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_I64:
                *SInt64p++ += 1; if (isComplex) *SInt64p++ += 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_I64 | NT_UNSIGNED:
                *UInt64p++ += 1; if (isComplex) *UInt64p++ += 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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.

Values returned for signed and unsigned 64-bit integer waves will be imprecise if the value exceeds  $2^{53}$  in magnitude because double-precision floating point can not precisely represent integers larger than  $2^{53}$ . See **64-bit Integer Issues** on page 215 for further discussion. If you must use 64-bit integer waves at full precision, use **MDGetNumericWavePointValueSInt64** or **MDGetNumericWavePointValueUInt64** instead.

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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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.

Values stored in signed and unsigned 64-bit integer waves will be imprecise if the value exceeds  $2^{53}$  in magnitude because double-precision floating point can not precisely represent integers larger than  $2^{53}$ . See **64-bit Integer Issues** on page 215 for further discussion. If you must use 64-bit integer waves at full precision, use **MDSetNumericWavePointValueSInt64** or **MDSetNumericWavePointValueUInt64** instead.

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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 for details.

## MDGetNumericWavePointValueSInt64

int

**MDGetNumericWavePointValueSInt64**(waveH, indices, value)

```
waveHndl waveH;           // Handle to the wave of interest
IndexInt indices[MAX_DIMENSIONS]; // Identifies the point of interest
SInt64 value[2];          // Value returned here
```

Returns via value the value of a particular element in the specified numeric wave. The value returned is always signed 64-bit integer, regardless of the data type of the wave.

This routine can be called for any numeric wave but is intended for use with signed 64-bit integer (NT\_I64) waves. For most applications, use **MDGetNumericWavePointValue** instead.

Values returned for floating point waves are truncated to integers. Values returned will be incorrect if the wave value is outside the range of signed 64-bit integers (-9223372036854775808 to 9223372036854775807). These issues do not apply if the data type of the specified wave is NT\_I64. See **64-bit Integer Issues** on page 215 for further discussion.

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.

MDGetNumericWavePointValueSInt64 was added in Igor Pro 7.03. If you call this with an earlier version of Igor, it will return IGOR\_OBSOLETE and do nothing.

### *Thread Safety*

MDGetNumericWavePointValueSInt64 is Igor-thread-safe with Igor Pro 7.03 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 for details.

## MDSetNumericWavePointValueSInt64

int

**MDSetNumericWavePointValueSInt64**(waveH, indices, value)

```
waveHndl waveH;           // Handle to the wave of interest
IndexInt indices[MAX_DIMENSIONS]; // Identifies the point of interest
SInt64 value[2];          // Value passed here
```

Sets the value of a particular point in the specified numeric wave. The value that you supply is always signed 64-bit integer, regardless of the data type of the wave.

This routine can be called for any numeric wave but is intended for use with signed 64-bit integer (NT\_I64) waves. For most applications, use **MDSetNumericWavePointValue** instead.

Values stored in floating point waves will be imprecise if the value exceeds the range of integers that can be represented precisely in the wave's data type. Values stored in integer waves will be incorrect if value is



outside the range that can be represented by the wave's data type. These issues do not apply if the data type of the specified wave is NT\_I64. See **64-bit Integer Issues** on page 215 for further discussion.

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, MDSetNumericWavePointValueSInt64 ignores value[1].

The function result is 0 or an error code.

MDSetNumericWavePointValueSInt64 was added in Igor Pro 7.03. If you call this with an earlier version of Igor, it will return IGOR\_OBSOLETE and do nothing.

### *Thread Safety*

MDSetNumericWavePointValueSInt64 is Igor-thread-safe with Igor Pro 7.03 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 for details.

## MDGetNumericWavePointValueUInt64

int

**MDGetNumericWavePointValueUInt64**(waveH, indices, value)

```

waveHndl waveH;           // Handle to the wave of interest
IndexInt indices[MAX_DIMENSIONS]; // Identifies the point of interest
UInt64 value[2];          // Value returned here

```

Returns via value the value of a particular element in the specified numeric wave. The value returned is always unsigned 64-bit integer, regardless of the data type of the wave.

This routine can be called for any numeric wave but is intended for use with unsigned 64-bit integer (NT\_I64 | NT\_UNSIGNED) waves. For most applications, use **MDGetNumericWavePointValue** instead.

Values returned for floating point waves are truncated to integers. Values returned will be incorrect if the wave value is outside the range of unsigned 64-bit integers (0 to 18446744073709551615). These issues do not apply if the data type of the specified wave is NT\_I64 | NT\_UNSIGNED. See **64-bit Integer Issues** on page 215 for further discussion.

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.

MDGetNumericWavePointValueUInt64 was added in Igor Pro 7.03. If you call this with an earlier version of Igor, it will return IGOR\_OBSOLETE and do nothing.

*Thread Safety*

MDGetNumericWavePointValueUInt64 is Igor-thread-safe with Igor Pro 7.03 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 for details.

## MDSetNumericWavePointValueUInt64

```
int
MDSetNumericWavePointValueUInt64 (waveH, indices, value)
waveHndl waveH;           // Handle to the wave of interest
IndexInt indices[MAX_DIMENSIONS]; // Identifies the point of interest
UInt64 value[2];          // Value passed here
```

Sets the value of a particular point in the specified numeric wave. The value that you supply is always unsigned 64-bit integer, regardless of the data type of the wave.

This routine can be called for any numeric wave but is intended for use with unsigned 64-bit integer (NT\_I64 | NT\_UNSIGNED) waves. For most applications, use **MDSetNumericWavePointValue** instead.

Values stored in floating point waves will be imprecise if the value exceeds the range of integers that can be represented precisely in the wave's data type. Values stored in integer waves will be incorrect if value is outside the range that can be represented by the wave's data type. These issues do not apply if the data type of the specified wave is NT\_I64 | NT\_UNSIGNED. See **64-bit Integer Issues** on page 215 for further discussion.

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, MDSetNumericWavePointValueUInt64 ignores value[1].

The function result is 0 or an error code.

MDSetNumericWavePointValueUInt64 was added in Igor Pro 7.03. If you call this with an earlier version of Igor, it will return IGOR\_OBSOLETE and do nothing.

*Thread Safety*

MDSetNumericWavePointValueUInt64 is Igor-thread-safe with Igor Pro 7.03 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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.

Values returned for signed and unsigned 64-bit integer waves will be imprecise if the value exceeds  $2^{53}$  in magnitude because double-precision floating point can not precisely represent integers larger than

2<sup>53</sup>. If you must use 64-bit integer waves and require full precision for very large values, you must use **MDGetNumericWavePointValueSInt64**, **MDGetNumericWavePointValueUInt64**, or the direct access method as described under **The Direct Access Method** on page 126. See **64-bit Integer Issues** on page 215 for further discussion.

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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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**.

Values stored for signed and unsigned 64-bit integer waves will be imprecise if the value exceeds 2<sup>53</sup> in magnitude because double-precision floating point can not precisely represent integers larger than 2<sup>53</sup>. If you must use 64-bit integer waves and require full precision for very large values, you must use **MDSetNumericWavePointValueSInt64**, **MDSetNumericWavePointValueUInt64**, or the direct access method as described under **The Direct Access Method** on page 126. See **64-bit Integer Issues** on page 215 for further discussion.

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*)WMNewPtr(numBytes);
if (dPtr==NULL)
    return NOMEM;

if (result = MDGetDPDataFromNumericWave(waveH, dPtr)) {
    WMDisposePtr((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)) {
    WMDisposePtr((Ptr)dPtr);
    return result;
}

WMDisposePtr((Ptr)dPtr);
```

*Thread Safety*

MDStoreDPDataInNumericWave is Igor-thread-safe with Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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.

Scaling of signed and unsigned 64-bit integer values is subject to inaccuracies for values exceeding  $2^{53}$  in magnitude because calculations are done in double-precision and double-precision can not precisely represent the full range of 64-integer values. Values returned for signed and unsigned 64-bit integer data will be imprecise if the value exceeds  $2^{53}$  in magnitude because double-precision floating point can not precisely represent integers larger than  $2^{53}$ . See **64-bit Integer Issues** on page 215 for further discussion.

The function result is zero or an error code.

### *Thread Safety*

FetchNumericValue is Igor-thread-safe with Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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].

Values stored for signed and unsigned 64-bit integer data will be imprecise if the value exceeds  $2^{53}$  in magnitude because double-precision floating point can not precisely represent integers larger than  $2^{53}$ . See **64-bit Integer Issues** on page 215 for further discussion.

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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 **WMGetHandleSize** 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 233.

See the example below for MDSetTextWavePointValue.

### *Thread Safety*

MDGetTextWavePointValue is Igor-thread-safe with Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 178 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 = WMNewHandle(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 (WMPtrAndHand("*", textH, 1)) { // Append an asterisk to handle
        result = NOMEM;
        break;
    }
    if (result = MDSetTextWavePointValue(waveH, indices, textH))
        break;
}
}

WMDisposeHandle(textH);

```

### *Thread Safety*

MDSetTextWavePointValue is Igor-thread-safe with Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 **WMDisposeHandle**.

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"&lt;null&gt; "One"&lt;null&gt; "Two"&lt;null&gt;</pre>
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>&lt;Offset to "Zero"&gt; &lt;Offset to "One"&gt; &lt;Offset to "Two"&gt; &lt;Extra offset&gt; "Zero"&lt;null&gt; "One"&lt;null&gt; "Two"&lt;null&gt;</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>&lt;Offset to "Zero"&gt; &lt;Offset to "One"&gt; &lt;Offset to "Two"&gt; &lt;Extra offset&gt; "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 is 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.

As explained under **WM Memory XOPSupport Routines** on page 195, on Macintosh the Handle data types are limited to 2 GB. If the wave text data requires more than 2 GB, `GetTextWaveData` will fail and return an error, even when running IGOR64.

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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 **WMDisposeHandle**.

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.

As explained under **WM Memory XOPSupport Routines** on page 195, on Macintosh the Handle data types are limited to 2 GB. If the wave text data requires more than 2 GB, you can not use SetTextWaveData, even when running IGOR64.

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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 131 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 131 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 Igor Pro 6.20 or later with certain exceptions explained under **Waves Passed to Igor Threads as Parameters** on page 177. 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 178 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 242.

To set file loader output variables, see **Routines for File-Loader XOPs** on page 345.

### 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 74).

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 <b>VAR_GLOBAL</b>	Creates global string variable
<b>NT_FP64</b> or ( <b>NT_FP64</b>   <b>VAR_GLOBAL</b> )	Creates global numeric variable
( <b>NT_FP64</b>   <b>NT_CMPLX</b> ) or ( <b>NT_FP64</b>   <b>NT_CMPLX</b>   <b>VAR_GLOBAL</b> )	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, Variable 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 74) to implement your external operation. Operation Handler will create output variables for you so you should not use Variable for that purpose.

Prior to Igor Pro 7, Variable allowed you to create a variable with a liberal name. This caused problems later because Igor does not support liberal names for variables. With Igor Pro 7 or later, Variable returns an error if the name is not a valid standard Igor name.

### 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 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 178 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 74).

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 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 178 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 74).

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 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 178 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 74).

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 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 178 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 74).

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 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 178 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 74).

Note that the text in the handle is *not* null terminated. Use **WMGetHandleSize** 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 233.

**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 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 178 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 74).

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, SetIgorIntVar 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 74) to implement your external operation. Operation Handler will create output variables for you so you should not use SetIgorIntVar for that purpose.

Prior to Igor Pro 7, SetIgorIntVar allowed you to create a variable with a liberal name. This caused problems later because Igor does not support liberal names for variables. With Igor Pro 7 or later, SetIgorIntVar returns an error if the name is not a valid standard Igor name.

### *Thread Safety*

SetIgorIntVar is Igor-thread-safe with 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 178 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 74).

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, `SetIgorFloatingVar` 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 74) to implement your external operation. Operation Handler will create output variables for you so you should not use `SetIgorFloatingVar` for that purpose.

Prior to Igor Pro 7, `SetIgorFloatingVar` allowed you to create a variable with a liberal name. This caused problems later because Igor does not support liberal names for variables. With Igor Pro 7 or later, `SetIgorFloatingVar` returns an error if the name is not a valid standard Igor name.

### *Thread Safety*

`SetIgorFloatingVar` is Igor-thread-safe with 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 178 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 74).

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, `SetIgorComplexVar` 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 74) to implement your external operation. Operation Handler will create output variables for you so you should not use `SetIgorComplexVar` for that purpose.

Prior to Igor Pro 7, `SetIgorComplexVar` allowed you to create a variable with a liberal name. This caused problems later because Igor does not support liberal names for variables. With Igor Pro 7 or later, `SetIgorComplexVar` returns an error if the name is not a valid standard Igor name.

*Thread Safety*

SetIgorComplexVar is Igor-thread-safe with 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 178 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 74).

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 74) 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 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 178 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 211.

### *Thread Safety*

GetNVAR is Igor-thread-safe with 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 178 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 211.

### *Thread Safety*

SetNVAR is Igor-thread-safe with 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 178 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.

svp is a pointer to an SVAR field in an Igor structure. The Igor structure pointer would be passed into the XOP as a parameter 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 **WMGetHandleSize** 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 233.

Returns 0 or an error code.

See also **NVARs and SVARs In Structures** on page 211.

### *Thread Safety*

GetSVAR is Igor-thread-safe with 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 178 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.

svp is a pointer to an SVAR field in an Igor structure. The Igor structure pointer would be passed into the XOP as a parameter 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 233.

Returns 0 or an error code.

See also **NVARs and SVARs In Structures** on page 211.

### *Thread Safety*

SetSVAR is Igor-thread-safe with 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 178 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 141 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:

<b>GetRootDataFolder</b>	<b>GetCurrentDataFolder</b>	<b>GetNamedDataFolder</b>
<b>GetDataFolderByIDNumber</b>	<b>GetParentDataFolder</b>	<b>GetIndexedChildDataFolder</b>
<b>GetDataFolderObject</b>	<b>GetWavesDataFolder</b>	<b>NewDataFolder</b>

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 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 178 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 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 178 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 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 178 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 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 178 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 **WMGetHandleSize** 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 233.

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 = WMNewHandle(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>
WMDisposeHandle(listingH);
```

### Thread Safety

GetDataFolderListing is Igor-thread-safe with 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 178 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 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 178 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 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 178 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 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 178 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 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 178 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 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 178 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()) {
    DataFolderHandle parentDFH;
    if (GetParentDataFolder(dfH, &parentDFH) == NO_PARENT_DATAFOLDER)
        dataFolderIsFree = 1;      // It's free
}
```

*Thread Safety*

GetParentDataFolder is Igor-thread-safe with 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 178 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 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 178 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 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 178 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 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 178 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 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 178 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 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 178 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 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 178 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 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 178 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 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 178 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 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 178 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.

Note that the text in a string handle is *not* null terminated. Use **WMGetHandleSize** 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 233.

The function result is 0 or error code.

*Thread Safety*

GetIndexedDataFolderObject is Igor-thread-safe with 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 178 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.

Use these handles immediately and do not store them. Once you return to Igor, Igor can kill the underlying objects.

Note that the text in a string handle is *not* null terminated. Use **WMGetHandleSize** 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 233.

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 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 178 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 **WMSetHandleSize** 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 233.

The function result is 0 or error code if there is no object with the specified name.

### Example

```
DataObjectValue v;
int objectType;
Handle strH = NULL;
double numVarValue;
DataFolderHandle dfH = NULL;
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 (WMHandToHand(&strH)) // Make our own copy.
            return NOMEM;
        if (WMPtrAndHand("***", strH, 3)) { // Append *** to string
            WMDisposeHandle(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);
WMDisposeHandle(strH);         // Dispose our copy of handle. (OK if NULL)
return err;
```

### Thread Safety

SetDataFolderObject is Igor-thread-safe with 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 178 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 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 178 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 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 178 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 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 178 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 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 178 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 143 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 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 178 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 143 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 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 178 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.

The old routines were removed from XOP Toolkit 7. If you are updating an existing XOP see **XOP Menus in XOP Toolkit 7** on page 421.

### 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 150 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 150 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 150 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 150 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 (IsIgorWindowActive((xopWindowRef)) // Is XOP window active?
    SetIgorMenuItem(kXOPWindowMessageCopy, 1, NULL, 0);
```

kXOPWindowMessageCopy 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 (IsIgorWindowActive((xopWindowRef)) // Is XOP window active?
    SetIgorMenuItem(kXOPWindowMessageUndo, 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 currently unused - pass 0 for it.

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.

Added in Igor Pro 6.40 but works with any version.

### *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.

Added in Igor Pro 6.40 but works with any version.

### *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.

Added in Igor Pro 6.40 but works with any version.

### *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.

Added in Igor Pro 6.40 but works with any version.

### *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.

Returns 0 or an error code.

Added in Igor Pro 6.40 but works with any version.

### *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.

Added in Igor Pro 6.40 but works with any version.

*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.

Added in Igor Pro 6.40 but works with any version.

*Thread Safety*

XOPFillWinMenu is not thread-safe.



## Routines for XOPs that Have Dialogs

Most dialog-related XOPSupport routines were removed in XOP Toolkit 7. If you are porting an old XOP that implements a dialog to XOP Toolkit 7 or later, see **XOP Dialogs in XOP Toolkit 7** on page 423 for details.

For background information on adding dialogs in an XOP, see **Adding Dialogs** on page 207.

### 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.

## 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 204.

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 205 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 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 178 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 205 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 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 178 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 MacOS 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.

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.

It is allowed for hfsPath and posixPath to point to the same memory.

When running with Igor Pro 6, hfsPath is assumed to be encoded as system text encoding as set by the preferred language in the Languages&Text control panel.

When running with Igor Pro 7 or later, hfsPath is assumed to be encoded as UTF-8.

posixPath is always encoded as UTF-8.

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 205 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 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 178 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 205 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 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 178 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 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 178 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 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 178 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 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 178 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 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 178 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 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 178 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 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 178 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 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 178 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 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 178 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
```

If your XOP requires Igor Pro 7.00 or later, use **XOPOpenFileDialog2** instead of XOPOpenFileDialog.

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.

## XOPOpenFileDialog2

```
int
XOPOpenFileDialog2(flagsIn, prompt, fileFilterStr, fileFilterIndexPtr,
                   initialDir, initialFile, flagsOutPtr, fullPathOut)
int flagsIn;                                     // Currently unused - must be 0
const char* prompt;                             // Prompt displayed in dialog
const char* fileFilterStr;
int* fileFilterIndexPtr;                         // May be NULL
const char* initialDir;                         // Full native path
const char* initialFile;
char fullPathOut[MAX_PATH_LEN+1]; // Set to a full native path
```

Displays the Open File dialog.

`XOPOpenFileDialog2` requires Igor Pro 7.00 or later. With earlier versions, it returns an error.

`XOPOpenFileDialog2` 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.

It returns the full path to the chosen file via `fullPathOut`. This is a native path, using colons on Macintosh, backslashes on Windows.

In the event of a cancel, `fullPathOut` is unmodified.

`flagsIn` is currently unused. Pass 0 for this parameter.

`prompt` sets the dialog window title.

`fileFilterStr` controls the filters in the file filter popup menu that appears in the Open File dialog. The construction of this string is the same as documented in the Igor help for the Open operation /F flag. For example, this populate the filter popup menu with three filters:

```
const char* fileFilters = "Data Files (*.txt,*.dat,*.csv):.txt,.dat,.csv;" \
                          "HTML Files (*.htm,*.html):.htm,.html;" \
                          "All Files:.*;";
```

`fileFilterIndexPtr` is ignored if it is NULL. If it is not NULL, then `*fileFilterIndexPtr` is the 1-based index of the file type filter to be initially selected. In the example given above, setting `*fileFilterIndexPtr` to 2 would select the "HTML Files" file filter on entry to the dialog. On exit from the dialog, `*fileFilterIndexPtr` is set to the 1-based index of the file filter string that the user last selected.

`initialDir` can be "" or it can point to a full native path to a directory, with or without the trailing path separator. It determines the directory that will be initially displayed in the Open File dialog. If "", the directory will be the last directory that was seen in the open or save file dialogs. `initialDir` is a native path using colons on Macintosh, backslashes on Windows.

initialFile can be "" or it can be the name to which the File Name edit control in the Save File dialog is to be initialized. initialFile works on Windows only because the Macintosh Open File dialog has no File Name edit control.

flagsOutPtr is currently unused. Pass NULL for this parameter.

XOPOpenFileDialog2 returns, via fullPathOut, the full path to the file that the user chose. If the user cancels, fullPathOut is unchanged. fullPathOut is a native path using colons on Macintosh, backslashes on Windows.

### *Thread Safety*

XOPOpenFileDialog2 is not thread-safe.

## XOPSaveFileDialog

```
int
XOPSaveFileDialog(prompt, fileFilterStr, fileIndexPtr, initDir,
                  defaultExtensionStr, fullPath)

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 fullPath[MAX_PATH_LEN+1]; // Output path returned here
```

If your XOP requires Igor Pro 7.00 or later, use **XOPSaveFileDialog2** instead of XOPSaveFileDialog.

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 fullPath. fullPath is both an input and an output as explained below. In the event of a cancel, fullPath is unmodified.

fullPath 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.

## XOPSaveFileDialog2

```

int
XOPSaveFileDialog2 (flagsIn, prompt, fileFilterStr, fileFilterIndexPtr,
                    initialDir, initialFile, flagsOutPtr, fullPathOut)
int flagsIn;                // Currently unused - must be 0
const char* prompt;         // Prompt displayed in dialog
const char* fileFilterStr;
int* fileFilterIndexPtr;    // May be NULL
const char* initialDir;    // Full native path
const char* initialFile;
char fullPathOut[MAX_PATH_LEN+1]; // Set to a full native path

```

Displays the Save File dialog.

XOPSaveFileDialog2 requires Igor Pro 7.00 or later. With earlier versions, it returns an error.

XOPSaveFileDialog2 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.

flagsIn is currently unused. Pass 0 for this parameter.

prompt sets the dialog window title.

fileFilterStr controls the filters in the file filter popup menu that appears in the Save File dialog. The construction of this string is the same as documented in the Igor help for the Open operation /F flag. For example, this populate the filter popup menu with three filters:

```

const char* fileFilters = "Data Files (*.txt,*.dat,*.csv):.txt,.dat,.csv;" \
                          "HTML Files (*.htm,*.html):.htm,.html;" \
                          "All Files:.*;";

```

fileFilterIndexPtr is ignored if it is NULL. If it is not NULL, then \*fileFilterIndexPtr is the 1-based index of the file type filter to be initially selected. In the example given above, setting \*fileFilterIndexPtr to 2 would select the "HTML Files" file filter on entry to the dialog. On exit from the dialog, \*fileFilterIndexPtr is set to the 1-based index of the file filter string that the user last selected.

initialDir can be "" or it can point to a full native path to a directory, with or without the trailing path separator. It determines the directory that will be initially displayed in the Save File dialog. If "", the directory will be the last directory that was seen in the Open File or Save File dialogs. initialDir is a native path using colons on Macintosh, backslashes on Windows.

initialFile can be "" or it can be the name to which the File Name edit control in the Save File dialog is to be initialized.

flagsOutPtr is currently unused. Pass NULL for this parameter.

XOPSaveFileDialog2 returns, via fullPathOut, the full path to the file that the user chose. fullPathOut is a native path using colons on Macintosh, backslashes on Windows.

**Thread Safety**

XOPSaveFileDialog2 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 not thread-safe because it calls **GetPathInfo2** which is not thread-safe.

## 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 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 178 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 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 178 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 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 178 for details.

## Routines for XOPs with Windows

Documentation for routines supporting XOP windows can be found in the following XOPSupport files:

XOPWindows.c

XOPContainers.c

See Chapter 9, **Adding Windows**, for further information.

## 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. Documentation for routines supporting text windows can be found in the following XOPSupport file:

XOPTextUtilityWindows.c

See **TU ("Text Utility") Windows** on page 174, for further information.



## Routines for Dealing with Resources

These callbacks are useful for accessing resources from within an XOP. Most XOPs will not need to deal directly with resources and so will not need to use these routines.

### 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.

There is additional documentation for FIFOs in the Igor Pro help; see the "FIFOs and Charts" help topic. 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 GBLoadWaveX, 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
SInt64 (64 bits)	SIGNED_INT, 8	NT_I64
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
UInt64 (64 bits)	UNSIGNED_INT, 8	NT_I64   NT_UNSIGNED
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;  // Number of values to convert
int srcBytes;        // Number of bytes in a point of input (1, 2, 4, or 8)
int srcFormat;       // XOP Toolkit code for input data format
int destBytes;       // Number 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 = 8    // 64 bit signed integer
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 = 8  // 64 bit unsigned 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, SInt64, SInt32, Short, Byte, UInt64, 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_I64            // 64 bit signed integer
NT_I32            // 32 bit signed integer
NT_I16            // 16 bit signed integer
NT_I8             // 8 bit signed integer
NT_I64 | NT_UNSIGNED // 64 bit unsigned integer
NT_I32 | 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.

All calculations are done in double precision.

Scaling of signed and unsigned 64-bit integer values is subject to inaccuracies for values exceeding  $2^{53}$  in magnitude because calculations are done in double-precision and double-precision can not precisely represent the full range of 64-integer values. In evaluating the result, if any value exceeds  $2^{53}$  in magnitude, the result is undefined. See **64-bit Integer Issues** on page 215 for further discussion.

*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 values that can be represented by the data type except for signed and unsigned 64-bit integer. The largest integer value that can be precisely represented in double-precision floating point is  $2^{53} = 9007199254740992$ . For signed 64-bit integer (NT\_I64), the minimum is -9007199254740992 and the maximum is 9007199254740992. For unsigned 64-bit integer (NT\_I64 | NT\_UNSIGNED), the minimum is 0 and the maximum is 9007199254740992.

If doRound is non-zero, the data is rounded to the nearest integer.

All calculations are done in double precision.

Scaling of signed and unsigned 64-bit integer values is subject to inaccuracies for values exceeding  $2^{53}$  in magnitude because calculations are done in double-precision and double-precision can not precisely represent the full range of 64-integer values. In evaluating the result, if any value exceeds  $2^{53}$  in magnitude, the result is undefined. See **64-bit Integer Issues** on page 215 for further discussion.

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)
const char* baseName;           // C string containing base name
char finalName[MAX_OBJ_NAME+1]; // 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 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 178 for details.

### UniqueName2

```
int
UniqueName2(nameSpaceCode, baseName, finalName, suffixNumPtr)
int nameSpaceCode;           // Usually MAIN_NAME_SPACE. See IgorXOP.h
const char* baseName;       // Base name to use for creating unique name
char finalName[MAX_OBJ_NAME+1]; // 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 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 178 for details.

## SanitizeWaveName

```
int
SanitizeWaveName(waveName, column)
char waveName[MAX_OBJ_NAME+1];    // Input and output wave name
int column;                        // Number of column being loaded or zero
```

This routine is obsolete. Use **CleanupName**, **CheckName** and **CreateValidDataObjectName** 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 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 178 for details.

## CleanupName

```
int
CleanupName(beLiberal, name, maxNameBytes)
int beLiberal;
char* name;
int maxNameBytes;
```

If your goal is to generate a valid object name that you can use to create a data object, consider using **CreateValidDataObjectName** instead of CleanupName.

CleanupName changes the name, if necessary, to make it a legal Igor object name.

On input, name may be longer than maxNameBytes bytes. On output, it is truncated if necessary.

For most uses, pass MAX\_OBJ\_NAME for the maxNameBytes parameter. name must be able to hold maxNameBytes+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 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 178 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 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 178 for details.

## CreateValidDataObjectName

```
int
CreateValidDataObjectName(
DataFolderHandle dataFolderH, // Handle to data folder or NULL
const char* inName,           // Input: Proposed name
char outName[MAX_OBJ_NAME+1], // 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 `GBLoadWaveX` 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 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 178 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 MAX\_OBJ\_NAME plus two additional bytes plus the null terminator. Thus you should declare name as shown above.

**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 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 178 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.

The memory pointed to by str must be big enough to hold an additional MAX\_OBJ\_NAME+2 bytes, plus the null terminator.

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[MAX_OBJ_NAME+2+32+1];
strcpy(cmd, "Display ");
CatPossiblyQuotedName(cmd, waveName);
XOPSilentCommand(cmd);
```

### *Thread Safety*

CatPossiblyQuotedName is Igor-thread-safe with 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 178 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.

### 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 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 178 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 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 178 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 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 178 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 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 178 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 **WMNewPtr**, and sets \*textPtrPtr to point to the allocated memory. You must dispose this when you no longer need it using **WMDisposePtr**.

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>
WMDisposePtr((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 **WMDisposeHandle**.

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 233.

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 = NULL;
int result;

if (result = HistoryFetchSelectedText(NULL, NULL, &h))
    return result;
<Deal with the text in handle>
WMDisposeHandle(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 **WMGetHandleSize** 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 233.

This handle belongs to you, so call **WMDisposeHandle** 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 **WMGetHandleSize** 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 233. 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 **WMDisposeHandle** 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 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 170 for further information.

#### *Thread Safety*

`SetIgorProcedure` is not thread-safe.

## DoWindowRecreationDialog

```
enum CloseWinAction
```

```
DoWindowRecreationDialog(procedureName)
```

```
char procedureName [MAX_OBJ_NAME+1];
```

This routine is used by very advanced XOPs that add a target window type to Igor. See **Adding XOP Target Windows** on page 170 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 212.

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 "ProcGlobal#MyFunction" or "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. If you want to call a function in the built-in ProcGlobal module, it is a good idea to specify it in the function name. See **CallFunction and Igor Independent Modules** on page 213 for an example.

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. A bug that prevented them from working was fixed in Igor Pro 7.01.

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 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 178 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 212.

`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 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 178 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 212.

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

Wave reference and DFREF parameters are supported in Igor Pro 6.10 or later. A bug that prevented them from working was fixed in Igor Pro 7.01.

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 89.

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 89 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 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 178 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 212.

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 208.

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. A bug that prevented them from working was fixed in Igor Pro 7.01.

**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 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 178 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 **WMGetHandleSize** 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 **WMGetHandleSize** 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 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 178 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 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 178 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 62.



*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 62.

*Thread Safety*

WindowsErrorToIgorError is thread-safe. It can be called from any thread.

## 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 67 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 67 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 **WMDisposeHandle**. \*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 67 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 67 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 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 178 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 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 178 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 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 178 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 **WMNewHandle**.

Note that the text in the handle is *not* null terminated. Use **WMGetHandleSize** 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 233. 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 using **WMNewHandle**.

Note that the text in the handle is *not* null terminated. Use **WMGetHandleSize** 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 233. 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 using **WMNewHandle**.

Note that the text in the handle is *not* null terminated. Use **WMGetHandleSize** 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 233. 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 thread-safe only when running with Igor Pro 8.00 or later.

## 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 using **WMNewHandle**. VariableList fills the handle with text.

Note that the text in the handle is *not* null terminated. Use **WMGetHandleSize** 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 233. 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 using **WMNewHandle**. StringList fills the handle with text.

Note that the text in the handle is *not* null terminated. Use **WMGetHandleSize** 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 233. 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 214 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 61 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 61 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

**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 67 for details.

See **Checking For Aborts** on page 214 for further discussion.

### *Thread Safety*

SpinProcess is Igor-thread-safe with 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 178 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>	"SimpleLoadWave XOP"
<subtopic name>	"SimpleLoadWave"
<topic name>[<subtopic name>]	"SimpleLoadWave XOP[SimpleLoadWave]"

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.

As of this writing, there is a problem in calling XOPDisplayHelpTopic from a Macintosh Cocoa XOP modal dialog running with Igor7. When the user dismisses the modal help dialog, the Cocoa XOP dialog is also dismissed. We currently have no solution for this problem.

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(xopWindowRef, msg, r)
IgorWindowRef xopWindowRef;
const char* message;           // The tip.
const Rect* r;                 // Hot rectangle.
```

message is a C string containing the message to display.

The function result is 0 if OK or IGOR\_OBSOLETE.

### *Igor7 or later*

Displays a message in the status bar.

On Macintosh, the status bar appears at the bottom of the command window. On Windows, it appears at the bottom of the MDI frame window.

The message is displayed until Igor is asked to display some other message, either by you via another call to XOPSetContextualHelpMessage, or by another XOP, or by an internal Igor routine.

The parameter r is ignored and can be NULL.

### *Igor6*

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.

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.

### *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
IgorWindowRef* windowRefPtr; // Pointer to IgorWindowRef
```

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.

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 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 233.

#### *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 233.

*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 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 178 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, strrchr2 is just like strrchr.

*Thread Safety*

strchr2 is Igor-thread-safe with 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 178 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.

## ConvertTextEncoding

```
int
ConvertTextEncoding(source, numSourceBytes, sourceTextEncoding, dest,
destBufSizeInBytes, destTextEncoding, errorMode, numOutputBytesPtr, options)
const char* source;           // Text to be converted
int numSourceBytes;           // Number of bytes in source text
WMTxtEncodingCode sourceTextEncoding; // Defined in IgorXOP.h
char* dest;                   // Output buffer
int destBufSizeInBytes;       // Size of output buffer
WMTxtEncodingCode destTextEncoding; // Defined in IgorXOP.h
WMTxtEncodingConversionErrorMode errorMode; // Defined in IgorXOP.h
WMTxtEncodingConversionOptions options; // Bitwise options
int* numOutputBytesPtr;       // Number of output bytes written
```

Converts text from one text encoding to another.

See **Text Encodings** on page 201 for background information about text encodings and how they are handled in Igor.

This routine uses ints for numSourceBytes and destBufSizeInBytes and thus the input text and the output buffer size must not exceed roughly 2 billion bytes.

source points to the input text for the conversion.

numSourceBytes is the number of bytes of source text.

sourceTextEncoding is a value of type WMTextEncodingCode - an enum defined in IgorXOP.h.

dest is the output buffer where converted data is stored. dest and source must not point to the same memory.

destBufSizeInBytes is the size of the output buffer in bytes. The number of bytes required depends on sourceTextEncoding and destTextEncoding. If destTextEncoding is kWMTTextEncodingUTF8 then the output buffer, in the worst case, needs to be 6\*numSourceBytes if errorMode is kWMTTextEncodingConversionErrorModeEscape or 4\*numSourceBytes otherwise. This is because a single input character requires, in the worst case, up to 4 bytes in UTF-8. Also if errorMode is kWMTTextEncodingConversionErrorModeEscape then the conversion will replace an unmappable input character with an escape sequence consisting of up to 6 bytes. The simplest solution is to always allocate an output buffer of size 6\*numSourceBytes. If you are converting very large chunks of source text you should use a smaller output buffer size chosen based on sourceTextEncoding and destTextEncoding.

destTextEncoding is a value of type WMTextEncodingCode - an enum defined in IgorXOP.h.

errorMode is a value defined by the WMTextEncodingConversionErrorMode type - an enum defined in IgorXOP.h. errorMode determines what happens if the source text can not be mapped to text in the text encoding specified by destTextEncoding. See the WMTextEncodingConversionErrorMode definition for details. For most purposes you should pass kWMTTextEncodingConversionErrorModeFail for errorMode. This causes ConvertTextEncoding to return an error if the source text can not be mapped to the destination text encoding.

The number of bytes in the converted output text is stored in \*numOutputBytesPtr.

options is a bitwise combination of WMTextEncodingConversionOptionFlag values as defined in IgorXOP.h. Typically you should form this parameter like this:

```
WMTextEncodingConversionOptions tecOptions = kTECOptionFlagDontAllowNulls;
tecOptions |= kTECOptionFlagDoConversionEvenIfEncodingsAreTheSame;
```

kTECOptionFlagDontAllowNulls causes ConvertTextEncoding to return an error if the input text contains nulls. Normally you don't want nulls in text but if for some reason you do, pass kTECOptionFlagAllowNulls instead of kTECOptionFlagDontAllowNulls.

kTECOptionFlagDoConversionEvenIfEncodingsAreTheSame forces ConvertTextEncoding to do the conversion even if sourceTextEncoding and destTextEncoding are the same. This is useful to validate that the source text is in fact valid text in the specified source text encoding.

If you know that the source text is valid, you can omit kTECOptionFlagDoConversionEvenIfEncodingsAreTheSame. In this case, if sourceTextEncoding and destTextEncoding are the same, ConvertTextEncoding merely copies the source text to the destination buffer without doing any validation.

The function result is 0 if OK or a non-zero error code.

Added in Igor Pro 7.00. If you call this when running with an earlier version it will return IGOR\_OBSOLETE and do nothing.

### ***Thread Safety***

ConvertTextEncoding is Igor-thread-safe with XOP Toolkit 7 and Igor Pro 7.00 or later. You can call it from a thread created by Igor but not from a private thread that you created yourself.

## 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.

## TickCount

```
TickCountInt
TickCount (void)
```

TickCount 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.

## 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 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 178 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 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 178 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.

## XOPGetClipboardData

```
int
XOPGetClipboardData(dataType, options, hPtr, lengthPtr)
OSType dataType;
int options;
Handle* hPtr;
BCInt* lengthPtr;
```

XOPGetClipboardData returns data from the clipboard in a handle via hPtr.

It returns the length of the data via lengthPtr.

dataType indicates the type of data you want to receive. The following values are supported:

'TEXT'	'UTF-8 text
'DIB '	'Windows only. The last character is a space.
'EMF '	'Windows only. The last character is a space.
'JPEG'	
'PDF '	The last character is a space.
'PNGf'	
'SVG '	The last character is a space.
'TIFF'	

options is for possible future use. Pass 0.

The output data is returned via hPtr. In the event of an error, \*hPtr is set to NULL.

If \*hPtr is not NULL then it belongs to you and you must dispose it via WMDisposeHandle.

The length of the output data is returned via lengthPtr. This will be the same as WMGetHandleSize(\*hPtr);

If you just want to tell if a given type of data is available on the clipboard and don't want to actually get that data, pass NULL for hPtr. In this case, XOPGetClipboardData sets \*lengthPtr to the length of the data of the specified type or to zero if there is no such data.

XOPGetClipboardData returns 0 in the event of success or a non-zero Igor error code.

XOPGetClipboardData was added in Igor Pro 7.00. If you call it while running an earlier version of Igor, it returns IGOR\_OBSOLETE and does nothing.

### *Thread Safety*

XOPGetClipboardData is not thread-safe.

## XOPSetClipboardData

```
int
XOPSetClipboardData(dataType, options, data, length)
OSType dataType;
int options;
void* data;
BCInt length;
```

XOPSetClipboardData stores data in the clipboard.

dataType indicates the type of data you are storing. The following values are supported:

'TEXT'	'UTF-8 text
'DIB '	'Windows only. The last character is a space.

'EMF '            'Windows only. The last character is a space.  
 'JPEG'  
 'PDF '            The last character is a space.  
 'PNGf'  
 'SVG '            The last character is a space.  
 'TIFF'

options is a bitwise parameter defined as follows:

Bit	Meaning
Bit 0	If cleared, XOPSetClipboardData clears the clipboard before storing data. If set, it does not clear the clipboard.
All other bits	Reserved for future use and must be cleared.

In most cases, you should pass 0 for options. This causes XOPSetClipboardData to clear the clipboard and then store your data in it. If bit 0 is set, XOPSetClipboardData does not clear the clipboard and you may wind up with multiple types of data in the clipboard.

data points to the data to be stored in the clipboard.

length is the number of bytes to be stored in the clipboard.

XOPSetClipboardData returns 0 in the event of success or a non-zero Igor error code.

On Windows, setting the clipboard via multiple successive calls can cause errors because programs that monitor the clipboard open it to inspect its contents each time it is changed. This causes Windows to return an error when Igor tries to open the clipboard. Repeatedly calling XOPSetClipboardData to store multiple formats on the clipboard may trigger this problem on some systems on some systems and is not recommended.

XOPSetClipboardData was added in Igor Pro 7.00. If you call it while running an earlier version of Igor, it returns IGOR\_OBSOLETE and does nothing.

#### ***Thread Safety***

XOPSetClipboardData is not thread-safe.

## WM Memory Routines

The WM memory XOPSupport routines provide a mechanism by which Igor and XOPs can exchange data in a compatible way.

WM memory XOPSupport routines use two data types, `Ptr` and `Handle`. They both provide access to a block of memory allocated in the heap. `Ptr` is rarely used while `Handle` is commonly used, most often for holding variable-length string data.

See **WM Memory XOPSupport Routines** on page 195 for background information.

### WMNewHandle

```
Handle
WMNewHandle(size)
BCInt size;
```

`WMNewHandle` allocates a block of size bytes in the heap and returns a reference to that block.

In the event of an error, `WMNewHandle` returns `NULL`. This corresponds to a `NOMEM` error. For example:

```
Handle h;
h = WMNewHandle(100);
if (h == NULL)
    return NOMEM;
```

You must call **WMDisposeHandle** to free the block of memory allocated by `WMNewHandle`

#### *Thread Safety*

`WMNewHandle` is thread-safe. It can be called from any thread.

### WMGetHandleSize

```
BCInt
WMGetHandleSize(h)
Handle h;
```

`h` is a handle referencing a block of memory in the heap.

`WMGetHandleSize` 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 `WMGetHandleSize`.

#### *Thread Safety*

`WMGetHandleSize` is thread-safe. It can be called from any thread.

### WMSetHandleSize

```
int
WMSetHandleSize(h, size)
Handle h;
BCInt size
```

`h` is a handle referencing a block of memory in the heap.

`WMSetHandleSize` 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 198 for details.

`WMSetHandleSize` returns 0 if it succeeds or a non-zero error code if it fails.

*Example*

```

Handle h;
Ptr p;
int err = 0;

h = WMNewHandle(100);           // Allocate handle
if (h == NULL)
    return NOMEM;
p = *h;                         // Dereference handle
. . .                           // Work with handle data using p
err = WMSetHandleSize(h, 200); // This may relocate handle data in memory
if (err != 0) {
    WMDisposeHandle(h);
    return err;
}
p = *h;                         // Re-dereference handle
. . .                           // Work with handle data using p
WMDisposeHandle(h);             // All done

```

*Thread Safety*

WMSetHandleSize is thread-safe. It can be called from any thread.

**WMHandToHand**

```

int
WMHandToHand(destHandlePtr)
Handle* destHandlePtr;

```

WMHandToHand 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 = WMHandToHand(&h2))
    return err;
<Use h2>           // h2 now refers to a new block of memory.
WMDisposeHandle(h2);

```

*Thread Safety*

WMHandToHand is thread-safe. It can be called from any thread.

**WMHandAndHand**

```

int
WMHandAndHand(h1, h2)
Handle h1;
Handle h2;

```

h1 and h2 are handles created by **WMNewHandle**.

WMHandAndHand 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*

WMHandAndHand is thread-safe. It can be called from any thread.

## WMDisposeHandle

```
void
WMDisposeHandle(h)
Handle h;
```

h is a handle referencing a block of memory in the heap.

WMDisposeHandle frees the block of memory that h references. After calling WMDisposeHandle, h is no longer a valid reference.

*Thread Safety*

WMDisposeHandle is thread-safe. It can be called from any thread.

## WMNewPtr

```
Ptr
WMNewPtr(size)
BCInt size;
```

WMNewPtr allocates a block of size bytes in the heap and returns a pointer to that block.

In the event of an error, WMNewPtr returns NULL. This corresponds to a NOMEM error. For example:

```
Ptr p;
p = (Ptr)WMNewPtr(100);
if (p == NULL)
    return NOMEM;
```

You must call **WMDisposePtr** to free the block of memory allocated by WMNewPtr.

*Thread Safety*

WMNewPtr is thread-safe. It can be called from any thread.

## WMGetPtrSize

```
BCInt
WMGetPtrSize(p)
Ptr p;
```

p is a pointer to a block of memory allocated by **WMNewPtr**.

WMGetPtrSize returns the size in bytes of that block of memory.

*Thread Safety*

WMGetPtrSize is thread-safe. It can be called from any thread.

## WMSetPtrSize

```
int
WMSetPtrSize(p, size)
Ptr p;
BCInt size;
```

p is a pointer to a block of memory allocated by **WMNewPtr**.

WMSetPtrSize resizes the block to the number of bytes specified by the size parameter.

WMSetPtrSize returns 0 if it succeeds or a non-zero error code if it fails.

*Thread Safety*

WMSetPtrSize is thread-safe.

## WMPtrToHand

```
int
WMPtrToHand(srcPtr, destHandlePtr, size)
Ptr srcPtr;
Handle* destHandlePtr;
BCInt size;
```

WMPtrToHand 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*

WMPtrToHand is thread-safe. It can be called from any thread.

## WMPtrAndHand

```
int
WMPtrAndHand(p, h, size)
Ptr* p;
Handle h;
BCInt size;
```

h is a handle referencing a block of memory in the heap.

WMPtrAndHand 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 198 for details.

WMPtrAndHand returns 0 if the allocation succeeded or an error code.

*Thread Safety*

WMPtrAndHand is thread-safe. It can be called from any thread.

## WMDisposePtr

```
void
WMDisposePtr(p)
Ptr p;
```

p is a pointer to a block of memory allocated by **WMNewPtr**.

WMDisposePtr frees the block of memory. After calling WMDisposePtr, you must not access the memory.

*Thread Safety*

WMDisposePtr is thread-safe. It can be called from any thread.





# XOP Toolkit 7 Upgrade Notes

## Overview

These notes are for XOP programmers transitioning from XOP Toolkit 6 to XOP Toolkit 7.

If you are targeting Igor Pro 8.00 or later, and you are creating a new XOP or updating an XOP already updated using XOP Toolkit 7.01 or later, you should use XOP Toolkit 8. See **Choosing Which XOP Toolkit to Use** on page 5 for details.

Here are the main new aspects of XOP Toolkit 7:

- XOP Toolkit 7 supports Igor Pro 6.20 or later (with some exceptions) and Igor Pro 7
- Macintosh 64-bit XOPs must be modified and recompiled in order to work with Igor Pro 8
- XOP Toolkit 7 adds a new and cleaner interface for dealing with XOP menus
- Major changes were made in the protocol for XOPs adding windows to Igor requiring significant changes in such XOPs
- The XOP Toolkit 6 support for adding dialogs was removed from XOP Toolkit 7
- As relates to text encoding, this manual and the XOP Toolkit 7 sample projects are optimized for use with Igor7.

These and many other changes are explained below in this chapter.

XOP Toolkit 7 can, in general, create XOPs compatible with Igor Pro 6.20.

XOPs that add dialogs will require significant changes.

XOPs that add windows will require major changes and will not be compatible with Igor Pro 6.

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 7

XOP Toolkit 7 provides support and samples for creating XOPs using these development systems:

For MacOS XOPs: Apple's Xcode 7 through Xcode 10

For Windows XOPs: Microsoft's Visual C++ 2015 and 2017

Newer development systems will likely be able to open these projects without problems.

## Create a Copy of Your Existing XOPs

The support and sample files for XOP Toolkit 7 come in a folder named IgorXOPs7.

## Appendix A — XOP Toolkit 7 Upgrade Notes

---

If you have existing XOP projects, *copy* them into the IgorXOPs7 folder. Leave your old projects as they are and do all new development in the new folder.

In most cases you can easily support Igor6 (6.20 or later) and Igor7 with one XOP. An exception is if you make frequent use of non-ASCII text. In that case, you may be better off maintaining separate Igor6 and Igor7 versions of your XOP - see **Text Encodings** on page 201 for details.

### Changes You Must Make

If you have XOPs developed with XOP Toolkit 6, here are the changes that you must make in order to use XOP Toolkit 7. We recommend that you familiarize yourself with this entire appendix first and then come back to this section and perform each of these steps.

Some changes are required in your Xcode projects. See **Xcode Project Changes For XOP Toolkit 7** on page 427 for details.

Some changes are required in your Visual C++ projects. See **Visual C++ Project Changes For XOP Toolkit 7** on page 434 for details.

If your Macintosh XOP project is very old it may use the GCC compiler. You need to change it to use the LLVM compiler. This is explained under **Converting to the LLVM Compiler** on page 44.

If you use obsolete memory management calls, remove them. See **Memory Management Changes** on page 417 for details.

If your XOP treats wave handles as regular handles, you must not do this. See **Change to Wave Handles** on page 417 for details.

Macintosh 64-bit XOPs must be modified and recompiled in order to work with Igor Pro 8 (in development as of this writing in May of 2017). Even if your existing XOP is Windows-only or 32-bit only, we recommend that you modify and recompile it. See **WM Memory XOPSupport Routines** on page 195 for details.

Obsolete XOPSupport routines have been removed from XOP Toolkit 7. See the section **Removed XOPSupport Routines** on page 419 for the list of removed routines. In most cases 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 6.

Some changes to your source code may be required because of compiler changes. See **Fixes for Compiler Changes** on page 421.

XOP Toolkit 7 requires Igor Pro 6.20 or later. Make sure that your main function checks the version of Igor it is running with. See **Checking the Igor Version** on page 421 for details.

If your XOP uses the WaveNote XOPSupport routine you need to change it to use **WaveNoteCopy**. Carefully read the documentation for WaveNoteCopy before making this change.

If your XOP uses the XOPOpenFileDialog or XOPSaveFileDialog routines, see **Changed XOPSupport Routines** on page 419 for important information.

If your XOP adds menus to Igor it may require changes. See **XOP Menus in XOP Toolkit 7** for details.

Macintosh balloon help and Windows menu help are obsolete. See **Help Resources in XOP Toolkit 7** for details.

If your XOP adds dialogs to Igor it will require substantial changes. See **XOP Dialogs in XOP Toolkit 7** on page 423 for details.

If your XOP adds windows to Igor it will require substantial changes. See **XOP Windows in Igor Pro 7** on page 424 for details.

Build and test your XOP with XOP Toolkit 7. Examine all compiler warnings and determine if they are harmless or indicate a real problem. Eliminate compiler warnings where possible.

To allow your XOP to run with the 64-bit version of Igor, see **64-bit XOPs** on page 187.

Igor Pro 7 uses UTF-8 text encoding to store text data. If your XOP exchanges non-ASCII text with Igor it may require changes. See **Text Encodings Used By XOPSupport Routines** for details.

## Removed Sample XOPs

Because they are complex and used by very few XOP programmers, the WindowXOP1 and TUDemo sample XOPs which were shipped with XOP Toolkit 6 have been removed from XOP Toolkit 7.

If you are an advanced programmer and want to add a text utility window to your XOP, send a note to WaveMetrics support requesting the Igor7-compatible TUDemo sample XOP for XOP Toolkit 7.

If you are an advanced programmer and want to create windows in your XOP, see **XOP Windows in Igor Pro 7** on page 424.

## Changes from GBLoadWave to GBLoadWaveX

GBLoadWaveX is the descendent of the GBLoadWave XOP that was provided as a sample XOP in XOP Toolkit 1 through XOP Toolkit 6. In Igor7, GBLoadWave was built into Igor itself, so the sample XOP was renamed GBLoadWaveX.

The GBLoadWaveX XOP requires Igor7 or later because it uses a callback, **XOPOpenFileDialog2**, which was added in Igor Pro 7.

In GBLoadWaveX, the /I flag turns on interactive mode, which displays an Open File dialog, and also allows you to specify platform-specific Open File dialog filter strings. In GBLoadWave it just turns on interactive mode.

GBLoadWaveX adds the /FILT flag for specifying Open File dialog filter strings.

The obsolete /D, /F and /L flags from GBLoadWave were removed in GBLoadWaveX. Use /T to specify the input and output data types instead.

## Memory Management Changes

If your XOP uses any of the following routines, remove calls to them:

HLock, HUnlock, HGetState, HSetState, MoveHHi, MoveLockHandle, SetWaveStates

## Change to Wave Handles

In Igor Pro 6 and before wave handles were implemented as regular handles. On Macintosh, regular handles can reference no more than 2 GB. In Igor Pro 7, to support 64 bits on Macintosh, WaveMetrics changed the implementation of wave handles so they are no longer regular handles.

It is unlikely that your XOP passes a wave handle to a regular handle routine like WMGetHandleSize. But if you do that, you need to change it to use XOPSupport routines only for managing wave handles, as described under **Routines for Accessing Waves** on page 253.

For example, if you used WMGetHandleSize to determine the size of a wave handle, change your code to use **WaveMemorySize** instead.

### Changes to Igor Behavior

Igor6 sends the CLEANUP message before Igor kills all experiment data. Igor7 and later send it after.

Igor7 and later send the NEW message when Igor is about to quit. Igor6 does not send the NEW message at that time.

### New Features

XOP Toolkit 7 provides a new, cleaner interface for adding menus to Igor. See **XOP Menus in XOP Toolkit 7** for details.

XOP Toolkit 7 provides a new protocol for supporting XOP windows. See **XOP Windows in Igor Pro 7** on page 424 for details.

XOP Toolkit 7 provides support for the signed and unsigned 64-bit integer data types. In Igor Pro 7.00 or later, Igor can create waves with these types. The constants NT\_I64 was added to IgorXOP.h. It can be used with the **MakeWave**, **MDMakeWave**, and **ConvertData2** XOPSupport routines and can be returned by the **WaveType** XOPSupport routine.

### 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.

The XOP\_WINDOW\_REF #define was replaced with XOPWindowRef. See **XOP Windows in Igor Pro 7** on page 424 for details.

All window-related messages were changed. See **XOP Windows in Igor Pro 7** on page 424 for details.

### Organizational Changes

The XOPFiles.c file has been split into three files: XOPFiles.c, XOPFilesMac.c and XOPFilesWin.c.

The definitions of messages sent to windows were moved from XOP.h to XOPWindows.h. These definitions are needed only by XOPs that add windows to Igor.

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.

### Text Encodings Used By XOPSupport Routines

Unlike previous versions, Igor7 uses UTF-8, a form of Unicode, to represent text.

If your XOP does not exchange text data with Igor then this change will not affect you.

Since UTF-8 is a superset of ASCII, if your XOP exchanges only ASCII text with Igor then this change will not affect you.

If your XOP obtains non-ASCII text from an external source (e.g., a file) and sends it to Igor, you may need to convert the text from whatever text encoding it uses to UTF-8 before sending it.

If your XOP obtains non-ASCII text from Igor and sends it to an external destination (e.g., a file), you may need to convert the text from UTF-8 to the destination text encoding before sending it to the external destination.

If your XOP searches for non-ASCII characters in text obtained from Igor, you may need to change it.

In most cases you can easily support Igor6 (6.20 or later) and Igor7 with one XOP. An exception is if you make frequent use of non-ASCII text. In that case, you may be better off maintaining separate Igor6 and Igor7 versions of your XOP - see **Text Encodings** on page 201 for details.

## Removed XOPSupport Routines

The following table lists routines removed from XOP Toolkit 7 and replacement routines, if any.

Removed Routine	Replacement Routine
WaveNote	<b>WaveNoteCopy</b>
IBeamCursor	SetXOPContainerMouseCursor
HandCursor	SetXOPContainerMouseCursor
SpinCursor	<b>SpinProcess</b>
SendWinMessageToIgor	None

In addition many old menu-related routines were removed. See **XOP Menus in XOP Toolkit 7**.

Also many old window-related routines were removed. See **Removed Window-Related XOPSupport Routines** on page 173.

In addition, all dialog support routines were removed. See **XOP Dialogs in XOP Toolkit 7** on page 423 for details.

## Changed XOPSupport Routines

The behavior of these routines has changed in XOP Toolkit 7 or with Igor7:

New Routine	Description
TransformWindowCoordinates	Requires Igor7 or later.
<b>WinInfo</b>	In XOP Toolkit 7, this function takes a new IgorWindowRef* parameter.
ArrowCursor	Does nothing when running with Igor7 or later. This function is no longer needed because, if your operation takes a long time, the operating system will spin the beachball.
WatchCursor	Does nothing when running with Igor7 or later. This function is no longer needed because, if your operation takes a long time, the operating system will spin the beachball.

Also some Text Utility Window routines were changed. See **Changed Text Utility Window Callbacks** on page 174.

### New XOPSupport Routines

The following sections list new routines added in XOP Toolkit 7.:

New Routine	Description
<b>ConvertTextEncoding</b>	Converts text from one text encoding to another.
<b>WaveTextEncoding</b>	Returns the text encoding code for an element of a wave. Most XOPs will not need to use this routine.
<b>WaveNoteCopy</b>	Returns a handle containing a copy of the wave's note text. Replaces WaveNote routine.
<b>XOPOpenFileDialog2</b>	See <b>Using XOPOpenFileDialog2 Instead of XOPOpenFileDialog</b> on page 420.
<b>XOPSaveFileDialog2</b>	See <b>Using XOPSaveFileDialog2 instead of XOPSaveFileDialog</b> on page 420.
<b>XOPGetClipboardData</b>	Retrieves data from the clipboard.
<b>XOPSetClipboardData</b>	Stores data in the clipboard.

---

See also **XOP Menus in XOP Toolkit 7** on page 421.

Many new routines were added for XOPs that create windows. See **XOP Windows in Igor Pro 7** on page 424 for details.

#### Using XOPOpenFileDialog2 Instead of XOPOpenFileDialog

**XOPOpenFileDialog2** is a successor to the **XOPOpenFileDialog** routine. Unlike **XOPOpenFileDialog**, **XOPOpenFileDialog2** works in a platform-independent way.

**XOPOpenFileDialog2** was added in Igor Pro 7.00 so, if you use it, your XOP will require Igor Pro 7. If you are writing an XOP that requires Igor Pro 7, use **XOPOpenFileDialog2**. If your XOP must run with Igor Pro 6, use **XOPOpenFileDialog**.

**XOPOpenFileDialog2** has three additional parameters, relative to **XOPOpenFileDialog**, which you must add: **flagsIn**, **initialFile**, and **flagsOutPtr**. Also the **fileFilterStr** parameter, which previously used platform-specific syntax, now uses the same platform-independent syntax as the Igor Open/D/R operation. See **XOPOpenFileDialog2** on page 341 for details.

#### Using XOPSaveFileDialog2 instead of XOPSaveFileDialog

**XOPSaveFileDialog2** is a successor to the **XOPSaveFileDialog** routine. Unlike **XOPSaveFileDialog**, **XOPSaveFileDialog2** works in a platform-independent way.

**XOPSaveFileDialog2** was added in Igor Pro 7.00 so, if you use it, your XOP will require Igor Pro 7. If you are writing an XOP that requires Igor Pro 7, use **XOPSaveFileDialog2**. If your XOP must run with Igor Pro 6, use **XOPSaveFileDialog**.

**XOPSaveFileDialog2** has three additional parameters, relative to **XOPSaveFileDialog**, which you must add: **flagsIn**, **initialFile**, and **flagsOutPtr**. Also the **fileFilterStr** parameter, which previously used platform-specific syntax, now uses the same platform-independent syntax as the Igor Open/D/R operation. See **XOPSaveFileDialog2** on page 344 for details.

## Fixes for Compiler Changes

In Visual C++ 2015, Microsoft changed the C++ compiler to require a space between a literal string and a macro following it. See "String literals followed by macros" at <https://msdn.microsoft.com/en-us/library/Bb531344.aspx>.

This affects the use of the CR\_STR macro in the XOP Toolkit. For example, this:

```
XOPNotice("Test"CR_STR);
```

must be changed to this:

```
XOPNotice("Test" CR_STR);
```

to prevent errors when compiling with VC2015.

Search for all occurrences of CR\_STR in your XOP source code and add a space before the CR\_STR macro.

## Checking the Igor Version

XOP Toolkit 7 requires Igor Pro 6.20 or later. To prevent it from running with earlier versions of Igor, you must put a test in your main 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".

## XOP Menus in XOP Toolkit 7

Most XOPs either add no menus or menu items to Igor, or add just a single menu item to an existing Igor menu. Such XOPs do not need to use any XOP Toolkit menu support routines. Consequently for most XOPs, the issues discussed in this section are not relevant.

If your XOP Toolkit 6 XOP adds menus to Igor's main menu bar, shows and hides menus, adds and removes menu items, changes the text of menu items, enables and disables menu items, or checks and unchecks menu items, then you need to understand this material and modify your XOP accordingly. These changes provide compatibility with Igor Pro 7 and XOP Toolkit 7.

In Igor Pro 6 and before, support for XOP menus and menu items is based on the Mac OS 9 Menu Manager API and emulation of it provided on Windows by the IGOR.lib file.

In Igor Pro 7, the internal implementation of menus is completely changed. For compatibility with Igor7 a new set of menu XOPSupport routines was added to XOP Toolkit 6.40. The new routines, listed in the following table, are also supported by XOP Toolkit 7 and work with Igor Pro 6.00 and later. This means that an XOP written using the new routines is compatible with Igor6 and Igor7.

The new routines use a new type, XOPMenuRef, instead of MenuHandle.

The old routines have been removed from XOP Toolkit 7. If you are porting an XOP to XOP Toolkit 7, you need to replace calls to the old routines.

See the MenuXOP1 sample XOP for examples using the new routines.

If you run an old XOP - i.e., one that has not been updated to use the new menu XOPSupport routines added in XOP Toolkit 6.40 - and the old XOP adds menus or menu items, it will not work correctly with Igor7. This is because the MenuHandle used by old XOPs is not the menu handle used by Igor7. When you pass a MenuHandle to Igor as a parameter to a callback, Igor7 attempts to verify that you are passing a valid Igor7

## Appendix A — XOP Toolkit 7 Upgrade Notes

---

menu handle. This verification should prevent old XOPs that pass old MenuHandles back to Igor from crashing. But, to get the expected behavior from Igor7, you must update your XOP.

Here are the new routines:

New Routine	Replaces
<b>XOPActualMenuIDToMenuRef</b>	GetMenuHandle
<b>XOPResourceMenuIDToMenuRef</b>	ResourceMenuIDToMenuHandle
<b>XOPGetMenuInfo</b>	GetMenuID
<b>XOPCountMenuItems</b>	CountMItems
<b>XOPShowMainMenu</b>	WMInsertMenu followed by WMDrawMenuBar
<b>XOPHideMainMenu</b>	WMDeleteMenu followed by WMDrawMenuBar
<b>XOPGetMenuItemInfo</b>	Nothing - requires Igor Pro 6.32 or later
<b>XOPGetMenuItemText</b>	getmenuitemtext
<b>XOPSetMenuItemText</b>	setmenuitemtext
<b>XOPAppendMenuItem</b>	appendmenu
<b>XOPInsertMenuItem</b>	insertmenuitem
<b>XOPDeleteMenuItem</b>	DeleteMenuItem
<b>XOPDeleteMenuItemRange</b>	WMDeleteMenuItems
<b>XOPEnableMenuItem</b>	EnableItem
<b>XOPDisableMenuItem</b>	DisableItem
<b>XOPCheckMenuItem</b>	CheckItem
<b>XOPFillMenu</b>	FillMenu
<b>XOPFillMenuNoMeta</b>	FillMenuNoMeta
<b>XOPFillWaveMenu</b>	FillWaveMenu
<b>XOPFillPathMenu</b>	FillPathMenu
<b>XOPFillWinMenu</b>	FillWinMenu

The old calls use MenuHandles while the new calls use XOPMenuRefs.

A MenuHandle is a Mac OS type that is emulated on Windows by IGOR.lib. When running with Igor Pro 6 an XOPMenuRef is a MenuHandle but when running with Igor Pro 7 it is a different type defined internally in Igor7. Consequently your code must treat XOPMenuRef as an opaque type and must not assume that it is equivalent to a MenuHandle. Therefore you must not pass an XOPMenuRef to a Mac OS call or to an IGOR.lib call that emulates a Mac OS call and you must not pass a MenuHandle that you received from Mac OS to Igor as if it were a XOPMenuRef.

To guarantee that you observe these rules, change your code to use the new XOP menu API listed above and do not use any OS menu routines. Here is a procedure for converting an XOP to use the new API:

1. Change all occurrences of MenuHandle to XOPMenuRef.
2. Compile. Each use of a XOPMenuRef in an old call will generate an error.
3. Replace the old call with the corresponding new call.



## Help Resources in XOP Toolkit 7

Macintosh balloon help is obsolete. You should remove the following balloon help resources from your .r file:

hmnu                      hdlg                      STR# resources containing menu help strings

Windows status line help for menu items is obsolete. You should remove the following balloon help resources from your WinCustom.rc file:

HMNU                      MENUHELP                      STR# resources containing menu help strings

## XOP Dialogs in XOP Toolkit 7

Prior to XOP Toolkit 7, the XOPSupport library included cross-platform dialog support routines. These routines were used to implement dialogs such as the Load General Binary dialog in the GBLoadWave sample XOP.

On Macintosh, this dialog support was based on the Dialog Manager, which was part of the original Macintosh API in 1984. When Apple moved to MacOS, it continued to support the Dialog Manager via the Carbon library, so the XOP dialog support routines continued to work.

Although parts of Carbon still work, Apple has mostly deprecated or abandoned it in favor of the Cocoa API. Many parts of Carbon work in 32 bits only, not in 64 bits. As a result, the old XOP dialog support has hit the end of the road and was removed from the Macintosh and Windows XOPSupport libraries in XOP Toolkit 7.

As of 2015, in the era of MacOS 10.10 and Windows 10, XOPs that use the old XOP Toolkit dialog routines continue to work in 32 bits on Macintosh and in 32 and 64 bits on Windows. They do not work in 64 bits on Macintosh.

If you are updating an old XOP that uses the old XOP Toolkit dialog routines, one option is for you to stick with XOP Toolkit 6. The main downside to this is that there is no way to run in 64 bits on Macintosh. If you want to run in 64 bits on Macintosh, or if you must port to XOP Toolkit 7 for some other reason, you will need to rewrite your dialog code. How you do this depends on the platform you are targeting.

On Macintosh, you need to rewrite your dialog using Cocoa. The GBLoadWaveX sample XOP does this and provides a good starting point. You will, however, need to have some familiarity with Cocoa, which has a fairly steep learning curve for uninitiated programmers.

On Windows, you can rewrite your dialog using standard Windows programming techniques. Alternatively, you can continue to use your old code by adding the DialogUtilitiesWin.c and DialogUtilitiesWin.h files from the Extras/Win folder to your project. These files contain the old XOP dialog support code with a few minor changes. To use them, you must make the following changes:

Include the DialogUtilitiesWin.h file in your dialog source file (e.g., GBLoadWaveDialogWin.cpp).

In your dialog source file:

Replace all occurrences of XOP\_DIALOG\_REF with XOPDialogRef.

Replace all occurrences of GetRadBut with GetRadioButton.

Replace all occurrences of SetRadBut with SetRadioButton.

Change the last parameter of all calls to SetDDouble from a pointer to a double to a double.

### XOP Windows in Igor Pro 7

If you have an existing Igor Pro 6 XOP that adds a window to Igor and you want to port it to Igor Pro 7, see **Adding Windows** on page 155.

### TU ("Text Utility") Windows in Igor Pro 7

This section discusses changes required for existing XOPs that add text utility windows to Igor. You need to read this section only if you have an existing Igor6 window XOP and you want to port your XOP to Igor7. For further details on windows see **TU ("Text Utility") Windows** on page 174.

A TU window is a "text utility" window - a simple plain text editing window. The TUDemo sample XOP (not shipped with XOP Toolkit 7 but available on request) and VDT2 sample XOP illustrate the use of TU windows.

The ACTIVATE, UPDATE, CLICK, KEY, GROW, SETGROW and NULLEVENT window messages sent by Igor to the XOP have been replaced with new messages in Igor7. Here are the Igor7 window messages and the corresponding TU callbacks which you must call on receiving the message:

Igor7 Message	Callback / Notes
kXOPWindowMessageActivate	TUActivate(TUStuffHandle TU, int flag)  In Igor6 the second parameter to the TUActivate callback is was defined as the value of the modifiers field of the Macintosh EventRecord. In Igor7 it is 1 for an activate event and 0 for a deactivate event.
kXOPWindowMessageUpdate	TUUpdate(TUStuffHandle TU)
kXOPWindowMessageMouseClicked	TUClick(TUStuffHandle TU, WMMouseEventRecord* merP)  In Igor6 the second parameter to the TUClick callback is a pointer to a Macintosh EventRecord. In Igor7 it is a pointer to a WMMouseEventRecord.
kXOPWindowMessageMouseMove	TUNull(TUStuffHandle TU, WMMouseEventRecord* merP)  In Igor6 the second parameter to the TUNull callback is a pointer to a Macintosh EventRecord. In Igor7 it is a pointer to a WMMouseEventRecord.
kXOPWindowMessageKey	TUKey(TUStuffHandle TU, WMKeyboardEventRecord* eventPtr)  In Igor6 the second parameter to the TUKey callback is a pointer to a Macintosh EventRecord. In Igor7 it is a pointer to a WMKeyboardEventRecord.
kXOPWindowMessageResized	TUGrow(TUStuffHandle TU, int size)
kXOPWindowMessageSetSizeLimits	<none>
kXOPWindowMessageFind	TUFind(TUStuffHandle TU, int messageCode)  In Igor6 the second parameter to the TUFind callback is 1 for find, 2 for find same and 3 for find selection.  In Igor7 it is kXOPWindowMessageFind, kXOPWindowMessageFindSame, kXOPWindowMessageFindSameBackwards, kXOPWindowMessageFindSelection, kXOPWindowMessageFindSelectionBackwards, kXOPWindowMessageFindMultiple or kXOPWindowMessageUseSelectionForFind.

In Igor Pro 6 and before these window messages were handled internally by Igor on Windows. On Macintosh the XOP had to call back to the corresponding XOPSupport callback routine. As of Igor Pro 7 these messages are handled the same all platforms. The XOP receives the message and makes the callback shown above.



## Project Changes For XOP Toolkit 7

### Overview

This appendix describes change you need to make to your Xcode and Visual C++ projects when upgrading from XOP Toolkit 6 to XOP Toolkit 7.

If you are targeting Igor Pro 8.00 or later, and you are creating a new XOP or updating an XOP already updated using XOP Toolkit 7.01 or later, you should use XOP Toolkit 8. See **Choosing Which XOP Toolkit to Use** on page 5 for details.

### Xcode Project Changes For XOP Toolkit 7

XOP Toolkit 7 has been tested with Xcode 6 through Xcode 10.

Xcode 10 and later support 64-bit development only, not 32-bit development. Consequently it is quite different from Xcode 9. For that reason, Xcode 9 and Xcode 10 are handled in separate sections:

**Converting Your Older Xcode Project for Xcode 9** in the next section

**Converting Your Older Xcode Project for Xcode 10** on page 431

### Converting Your Older Xcode Project for Xcode 9

In this section we show how to update an Xcode 3, Xcode 4 or Xcode 6 project from XOP Toolkit 6 to an Xcode 9 project for use with XOP Toolkit 7. Make sure to back up your Xcode projects before starting this process.

We use XOP1 as an example. The XOP1 project that we are starting with is from XOP Toolkit 6.30. To convert your project, substitute its name for “XOP1” in the following instructions.

1. If you have not already done it, copy the project folder, XOP1, from your IgorXOPs6 folder to your IgorXOPs7 folder.  
All of the steps below are to be done on the copy in IgorXOPs7.
2. Open the XOP1 folder.
3. If you see multiple folders whose names begin with Xcode (e.g., Xcode, Xcode4, Xcode6), delete all but the highest numbered folder and rename the remaining folder as Xcode.  
At this point, you should have just one Xcode folder in the XOP1 folder.
4. Open XOP1/Xcode/XOP1.xcodeproj in Xcode 9.

### Updating the 32-bit Xcode Target

1. Choose File→Project Settings.  
Click the Advanced button.

## Appendix B — Project Changes For XOP Toolkit 7

---

For Build Location, choose Legacy.

Click Done.

Click Done again.

2. 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.

Click the XOP1 icon at the top of the Project Navigator pane. The project settings appear in the center pane.

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 XOP1 icon beneath it and TARGETS with another XOP1 icon beneath it in the left column of the center pane. This column is the Project and Targets List.

3. Click the XOP1 icon under the word PROJECT in the Project and Targets List.

Click the Info tab in the center pane.

From the "Mac OS Deployment Target" popup menu, select the earliest version of Mac OS that you want to support but it must be 10.9 or later.

4. Click the Build Settings tab.

Click the All and Combined buttons.

Under Architectures/Architectures, select 32-bit Intel.

Under Architectures/Base SDK, select Latest MacOS.

Under Build Options/Compiler for C/C++/Objective-C, select Default Compiler.

5. Click the XOP1 icon under the word TARGETS in the Project and Targets List.

Under Architectures/Architectures, select 32-bit Intel.

Under Architectures/Base SDK, select Latest MacOS.

Under Build Options/Compiler for C/C++/Objective-C, select Default Compiler.

Under Search Paths/Header Search Paths, enter "../XOPSupport" (without the quotes).

Under Search Paths/Library Search Paths, enter "../XOPSupport/Xcode" (without the quotes).

Under Apple LLVM Custom Compiler Flags, in the Other Warning Flags section, add

-Wno-parentheses

6. In the Project Navigator, select the libXOPSupport.a icon.

Choose View→Utilities→Show File Inspector to show the File Inspector pane on the right.

In the Identity and Type section, from the Location popup menu, choose Relative to Project.

Click the folder icon below the Location popup menu to display the Choose File dialog.

Navigate to the IgorXOPs7/XOPSupport/Xcode folder and select the libXOPSupport.a file.

Click the Choose button.

In the Target Membership section, make sure that XOP1 is checked.

7. Click the XOP1 icon at the top of the Project Navigator pane.

8. Click the External Frameworks and Libraries icon in the Project Navigator and choose File→Add Files to XOP1.

In the resulting Open File dialog, navigate to /System/Library/Frameworks and select AudioToolbox.framework (needed for the XOPBeep XOPSupport function).

Click the Add button.

Select the AudioToolbox.framework icon in the Project Navigator.

In the Target Membership section of the File Inspector pane, make sure that XOP1 is checked.

9. Click the External Frameworks and Libraries icon in the Project Navigator and choose File→Add Files to XOP1.

In the resulting Open File dialog, navigate to /System/Library/Frameworks and select Cocoa.framework (needed for the XOPOKAlert and similar XOPSupport functions).

Click the Add button.

Select the Cocoa.framework icon in the Project Navigator.

In the Target Membership section of the File Inspector pane, make sure that XOP1 is checked.

10. In the Project Navigator pane, select the XOP1.cpp file to display it in the main editor pane.

Scroll to the bottom of XOP1.cpp and look for the main or XOPMain function.

If you have an XOPMain function, skip to the next step.

You need to change your main function to XOPMain as this is required by the LLVM compiler now used by Xcode.

Change this:

```
HOST_IMPORT void  
main(IOWRecHandle ioRecHandle)
```

to this:

```
HOST_IMPORT int  
XOPMain(IOWRecHandle ioRecHandle)
```

Your XOP will require Igor Pro 6.20 or later. Update the code in the XOPMain function that checks the Igor version accordingly.

Save XOP1.cpp.

Select the XOP1.h icon.

Change this:

```
HOST_IMPORT void main(IOWRecHandle ioRecHandle);
```

to this:

```
HOST_IMPORT int XOPMain(IOWRecHandle ioRecHandle);
```

Save XOP1.h.

Open the Exports.exp file in the Xcode folder and change this:

```
_main
```

to this:

```
_XOPMain
```

Save Exports.exp.

11. Choose Product→Build. The XOP should be successfully built.

The compiled XOP, XOP1.xop, should appear in the IgorXOPs7/XOP1/Xcode/Debug folder.

You can also locate the compiled XOP by right-clicking XOP1.xop in the Products group of the Project Navigator pane and choosing Show In Finder.

12. Activate XOP1.xop as described under **Activating XOPs Using the Igor Extensions Folder** on page 6.

### Debugging the 32-bit Xcode XOP

1. Choose Product→Scheme→Edit Scheme.

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.

2. To start the debugger and launch Igor, choose Product→Run.

### Creating the 64-bit Xcode Target

Now we will add a 64-bit target to the project. On Macintosh, 64-bit XOPs require Igor Pro 7.00 or later.

1. Click the XOP1 icon at the top of the Project Navigator pane.

Click the XOP1 icon under the word TARGETS in the Project and Targets List.

Choose Edit→Duplicate. This creates a target named "XOP1 copy".

Choose Product→Scheme→Manage Schemes. In the resulting dialog, rename "XOP1 copy" as XOP1-64. (To rename the item you first have to click the name to select it. Then click again to enter rename mode. Edit the name and press Enter.)

(When you convert your own project, note that the name of 64-bit version of the XOP should contain a dash only if the original name ends with a digit. Thus the 64-bit version of XOP1 is XOP1-64, with a dash, but the 64-bit version of GBLoadWaveX is GBLoadWaveX64, without a dash.)

Click Close to exit the Manage Schemes dialog.

1. Click the "XOP1 copy" icon under the word TARGETS in the Project and Targets List.

Rename the target as XOP1-64.

2. Click the XOP1-64 icon under the word TARGETS in the Project and Targets List.

In the Build Settings tab, in Architectures/Architecture, select "64-bit Intel".

For Packaging/Info.plist File, enter Info64.plist.

For Packaging/Product Name, enter XOP1-64.

3. In the Project Navigator, rename the "XOP1 copy-Info.plist" icon, at the bottom of the navigator, as "Info64.plist".

Drag the Info64.plist icon into the Resources group.

In the Project Navigator pane, select the Info64.plist icon. In the Identity and Type section of the File Inspector, from the Location popup menu, choose Relative to Group.

In the Target Membership section of the File Inspector pane, make sure that XOP1 and XOP1-64 are unchecked.

4. Click the XOP1 icon at the top of the Project Navigator pane and click XOP1-64 under TARGETS in the Project and Targets List.

From the list of tabs click Info (just to the left of Build Settings).

In the center pane, change the Executable File setting to XOP1-64.

In the center pane, change the Bundle identifier from com.wavemetrics.xop.xop1 to com.wavemetrics.xop.xop1-64.

5. In the Project Navigator pane, click the External Frameworks and Libraries icon and choose File→Add Files to XOP1.

Navigate to the IgorXOPs7/XOPSupport/Xcode folder and select the libXOPSupport64.a file.

Click the Add button.

6. In the Project Navigator pane, click the libXOPSupport64.a icon.

In the Identity and Type section of the File Inspector pane, from the Location popup menu, choose Relative to Project.

Make sure that only XOP1-64 is checked in the Target Membership section.

7. In the Project Navigator pane, click the libXOPSupport.a icon.

In the Identity and Type section of the File Inspector pane, from the Location popup menu, choose Relative to Project.

Make sure that only XOP1 is checked in the Target Membership section.

8. Click the XOP1 icon at the top of the Project Navigator pane.

9. Choose Product→Scheme→XOP1-64 to make it the active scheme.



(Because of a bug in Xcode, if you now look at the Product→Scheme submenu, it will still say XOP1, not XOP1-64. The scheme popup menu near the top/left corner of the Xcode window should say “XOP-64 > My Mac”. If you don’t see the scheme popup menu, choose View→Show Toolbar.)

10. Choose Product→Build. The XOP should be successfully built.

The compiled XOP, XOP1-64.xop, should appear in the IgorXOPs7/XOP1/Xcode/Debug folder.

You can also locate the compiled XOP by right-clicking XOP1-64.xop in the Products group of the Project Navigator pane and choosing Show In Finder.

11. Activate XOP1-64.xop as described under **Activating XOPs Using the Igor Extensions Folder** on page 6.

### Debugging the 64-bit Xcode XOP

1. Choose Product→Scheme→Edit Scheme.

Select the XOP1-64 scheme from the Active Scheme popup menu in the Edit Scheme window.

Click the Run row 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 64-bit application, select it, and click Choose.

Click Close.

2. To start the debugger, choose Product→Run.

## Converting Your Older Xcode Project for Xcode 10

In this section we show how to update an Xcode 3, Xcode 4 or Xcode 6 project from XOP Toolkit 6 to an Xcode 10 project for use with XOP Toolkit 7. Make sure to back up your Xcode projects before starting this process.

Xcode 10 and later support 64-bit development only, not 32-bit development. If you need to maintain the 32-bit version of your XOP, you can not use Xcode 10.

We use XOP1 as an example. The XOP1 project that we are starting with is from XOP Toolkit 6.30. To convert your project, substitute its name for “XOP1” in the following instructions.

1. If you have not already done it, copy the project folder, XOP1, from your IgorXOPs6 folder to your IgorXOPs7 folder.  
All of the steps below are to be done on the copy in IgorXOPs7.
2. Open the XOP1 folder.
3. If you see multiple folders whose names begin with Xcode (e.g., Xcode, Xcode4, Xcode6), delete all but the highest numbered folder and rename the remaining folder as Xcode.  
At this point, you should have just one Xcode folder in the XOP1 folder.
4. Open XOP1/Xcode/XOP1.xcodeproj in Xcode 10.  
Xcode will display a dialog saying “32-bit MacOS targets are no longer supported”. Click OK.

### Converting the 32-bit Xcode Target to 64-bit

1. Choose File→Project Settings.

Click the Advanced button.

For Build Location, choose Legacy.

Click Done.

Click Done again.

## Appendix B — Project Changes For XOP Toolkit 7

---

2. 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. Click the XOP1 icon at the top of the Project Navigator pane. The project settings appear in the center pane. 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 XOP1 icon beneath it and TARGETS with another XOP1 icon beneath it in the left column of the center pane. This column is the Project and Targets List.
3. Choose Product→Scheme→Manage Schemes. Rename the XOP1 scheme as XOP1-64. (To rename the item you first have to click the name to select it. Then click again to enter rename mode. Edit the name and press Enter.) (When you convert your own project, note that the name of 64-bit version of the XOP should contain a dash only if the original name ends with a digit. Thus the 64-bit version of XOP1 is XOP1-64, with a dash, but the 64-bit version of GBLoadWaveX is GBLoadWaveX64, without a dash.) Click the Close button.
4. Click the XOP1 icon under the word TARGET in the Project and Targets List. Rename the target as XOP1-64.
5. Click the XOP1 icon under the word PROJECT in the Project and Targets List. Click the Info tab in the center pane. From the "Mac OS Deployment Target" popup menu, select the earliest version of Mac OS that you want to support but it must be 10.11 or later.
6. Click XOP1 under the word TARGETS. Click General (to the left of Info) to show the General tab. Make sure that the Deployment Target for the XOP1 target is the same as what you set for the XOP1 project.
7. Click the XOP1 icon under the word PROJECT in the Project and Targets List. Click the Build Settings tab. Click the All and Combined buttons. Under Architectures/Architectures, select "Standard Architectures (64-bit Intel)". Under Architectures/Base SDK, select MacOS. Under Build Options/Compiler for C/C++/Objective-C, select Default Compiler. Under Search Paths, set Always Search User Paths to No.
8. Click the XOP1 icon under the word TARGETS in the Project and Targets List. Under Architectures/Architectures, select "Standard Architectures (64-bit Intel)". Under Architectures/Base SDK, select MacOS. Under Build Options/Compiler for C/C++/Objective-C, select Default Compiler. For Packaging/Info.plist File, enter Info64.plist. For Packaging/Product Name, enter XOP1-64. Under Search Paths/Header Search Paths, enter "../XOPSupport" (without the quotes). Under Search Paths/Library Search Paths, enter "../XOPSupport/Xcode" (without the quotes). Under Search Paths, set Always Search User Paths to No. Under Apple Clang Custom Compiler Flags, in the Other Warning Flags section, add -Wno-parentheses
9. In the Project Navigator, rename the "Info.plist" icon as "Info64.plist".

10. Click the XOP1 icon in the Project Navigator and then click the XOP1 icon under the word TARGET in the Project and Targets List.

From the list of tabs click Info (just to the left of Build Settings).

In the center pane, change the Bundle identifier from `com.wavemetrics.xop.xop1` to `com.wavemetrics.xop.xop1-64`.

In the center pane, change the Executable File setting to XOP1-64.

11. In the Project Navigator pane, right-click the `libXOPSupport.a` icon and choose Delete.

If Xcode displays a dialog asking what you want to delete, click Delete Reference.

12. In the Project Navigator pane, click the External Frameworks and Libraries icon and choose File→Add Files to XOP1.

Navigate to the `IgorXOPs7/XOPSupport/Xcode` folder and select the `libXOPSupport64.a` file.

Click the Add button.

13. Choose View→Inspectors→Show File Inspector to show the File Inspector pane on the right.

14. In the Project Navigator pane, click the `libXOPSupport64.a` icon.

In the Identity and Type section of the File Inspector pane, from the Location popup menu, choose Relative to Project.

Make sure that XOP1-64 is checked in the Target Membership section.

15. Click the XOP1 icon at the top of the Project Navigator pane.

16. Click the External Frameworks and Libraries icon in the Project Navigator and choose File→Add Files to XOP1.

In the resulting Open File dialog, navigate to `/System/Library/Frameworks` and select `AudioToolbox.framework` (needed for the XOPBeep XOPSupport function).

Click the Add button.

Select the `AudioToolbox.framework` icon in the Project Navigator.

In the Target Membership section of the File Inspector pane, make sure that XOP1-64 is checked.

17. Click the External Frameworks and Libraries icon in the Project Navigator and choose File→Add Files to XOP1.

In the resulting Open File dialog, navigate to `/System/Library/Frameworks` and select `Cocoa.framework` (needed for the XOPOKAlert and similar XOPSupport functions).

Click the Add button.

Select the `Cocoa.framework` icon in the Project Navigator.

In the Target Membership section of the File Inspector pane, make sure that XOP1-64 is checked.

18. Choose View→Navigators→Show Issue Navigator.

You should see an item in the Issue Navigator pane that says "Update to recommended settings". Click it. A window will open that lists settings to be updated.

Click Perform Changes.

19. Choose View→Navigators→Show Project Navigator.

In the Project Navigator pane, select the `XOP1.cpp` file to display it in the main editor pane.

Scroll to the bottom of `XOP1.cpp` and look for the `main` or `XOPMain` function.

If you have an `XOPMain` function, skip to the next step.

You need to change your `main` function to `XOPMain` as this is required by the LLVM compiler now used by Xcode.

Change this:

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

to this:

## Appendix B — Project Changes For XOP Toolkit 7

---

```
HOST_IMPORT int  
XOPMain(IORecHandle ioRecHandle)
```

Your XOP will require Igor Pro 7.00 or later because there is no 64-bit version of Igor6. Update the code in the XOPMain function that checks the Igor version accordingly.

Save XOP1.cpp.

Select the XOP1.h icon.

Change this:

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

to this:

```
HOST_IMPORT int XOPMain(IORecHandle ioRecHandle);
```

Save XOP1.h.

Open the Exports.exp file in the Xcode folder and change this:

```
_main
```

to this:

```
_XOPMain
```

Save Exports.exp.

20. Choose Product→Build. The XOP should be successfully built.

The compiled XOP, XOP1-64.xop, should appear in the IgorXOPs7/XOP1/Xcode/Debug folder.

You can also locate the compiled XOP by right-clicking XOP1-64.xop in the Products group of the Project Navigator pane and choosing Show In Finder.

21. Activate XOP1-64.xop as described under **Activating XOPs Using the Igor Extensions Folder** on page 6.

### Debugging the 64-bit Xcode XOP

1. Choose Product→Scheme→Edit Scheme.

Click the Run row 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 64-bit application, select it, and click Choose.

Click Close.

2. To start the debugger, choose Product→Run.

## Visual C++ Project Changes For XOP Toolkit 7

XOP Toolkit 7 was created with Visual C++ 2015 and has also been tested with Visual C++ 2017. It will most likely work with later versions of Visual C++ and may work with earlier versions. However, because of compatibility issues, described under **Visual C++ Compatibility Issues** on page 47, you may run into problems mixing Visual C++ versions. If you use a different version of Visual C++, you must recompile the XOP Toolkit with your Visual C++ version.

### Converting Your Older Visual C++ Project

In this section we show how to update a Visual C++ project from XOP Toolkit 6 to a Visual C++ 2015 project for use with XOP Toolkit 7.

We use XOP1 as an example. The XOP1 project that we are starting with is from XOP Toolkit 6.30. To convert your project, substitute its name for “XOP1” in the following instructions.

1. If you have not already done it, copy the project folder, XOP1, from your IgorXOPs6 folder to your IgorXOPs7 folder.  
All of the steps below are to be done on the copy in IgorXOPs7.
2. Open the XOP1 folder.
3. If you see multiple folders whose names begin with VC (e.g., VC9, VC10, VC2012), delete all but the highest numbered folder and rename the remaining folder as VC.  
At this point, you should have just one VC folder in the XOP1 folder.
4. Open XOP1/VC/XOP1.sln in Visual C++ 2015.

### Updating the 32-bit Visual C++ Target

1. Choose View→Solution Explorer.
2. Right-click the XOP1 icon and choose Properties to display the XOP1 Property Pages dialog.  
From the Configuration popup menu, choose All Configurations.  
From the Platform popup menu, choose Win32.
3. Under Configuration Properties, select General and enter the following settings:  
*Output Directory:* .\  
*Target Name:* XOP1  
*Target Extension:* .xop  
*Character Set:* Use Multi-Byte Character Set  
Click Apply.
4. Under Configuration Properties, select Debugging and enter the following settings:  
*Command:* <path to your Igor.exe application>  
Click Apply.
5. Under C/C++, select General and enter the following settings:  
*Additional Include Directories:* ..\..\XOPSupport;%(AdditionalIncludeDirectories)  
*Debug Information Format:* Program Database (/Zi)  
Click Apply.
6. Under C/C++, select Command Line and add the following setting:  
*Additional Options:* /utf-8  
Click Apply.
7. Under C/C++, select Preprocessor.  
From the Configuration popup menu, choose Debug.  
Verify that `_CRT_SECURE_NO_WARNINGS` is included as a preprocessor definition. If not, add it.  
Click Apply.  
From the Configuration popup menu, choose Release.  
Verify that `_CRT_SECURE_NO_WARNINGS` is included as a preprocessor definition. If not, add it.  
Click Apply.
8. Under C/C++, select Code Generation.  
From the Configuration popup menu, choose Debug and enter the following settings:  
*Runtime Library:* Multi-Threaded Debug (/MTd)  
From the Configuration popup menu, choose Release and enter the following settings:

*Runtime Library:* Multi-Threaded (/MT)

From the Configuration popup menu, choose All Configurations and enter the following settings:

*Struct Member Alignment:* Default

Click Apply.

9. Under Linker, select General and enter the following settings:

*Output File:* XOP1.xop

Click Apply.

10. Under Linker, select Input and enter the following settings:

*Additional Dependencies:* version.lib;%(AdditionalDependencies)

Click Apply.

11. Under Linker, select System and enter the following settings:

*Enable Large Addresses:* Yes (/LARGEADDRESSAWARE)

Click Apply.

12. Under Linker, select Advanced and enter the following settings:

*Target Machine:* MachineX86

Click Apply.

13. Under Resources, select General and enter the following settings:

*Additional Include Directories:* ..\..\XOPSupport;%(AdditionalIncludeDirectories)

Click Apply.

14. Click OK to close the Property Pages dialog.

15. In the Solution Explorer, select the XOPSupport.lib icon.

Choose View→Properties Window.

In the Properties window, set Relative Path to: ..\..\XOPSupport\VC\XOPSupport.lib

Press Enter to enter the setting.

Close the Properties window.

16. Choose File→Save All.

17. Choose Build→Build Solution.

The project should successfully compile and link, producing XOP1.xop. You may get some warnings.

### Adding the 64-bit Visual C++ Target

1. Choose View→Solution Explorer.

In the Solution Explorer window, select the XOP1 icon.

Right-click the XOP1 icon and choose Properties to open the XOP1 Property Pages dialog.

2. Click the Configuration Manager button to open the Configuration Manager dialog.

Click the Active Solution Platform popup menu.

If you see x64 listed, your project already has a 64-bit platform. If so, close the Configuration Manager dialog and the XOP1 Property Pages dialog and skip to **Updating the 64-bit Visual C++ Target** on page 437.

3. Click the menu under Active Solution Platform and choose New to display the New Solution Platform dialog.

Enter x64 in the “Type or Select the New Platform” edit box.

From the Copy Settings From popup menu, choose Win32.

Check the Create New Project Platforms checkbox.

Click OK to close the New Solution Platform dialog.

Click Close to close the Configuration Manager dialog.

Click OK to close the XOP1 Property Pages dialog.

Choose File→Save All.

4. In the Solution Explorer window, select the XOP1 icon.  
Choose Project→Add Existing Item.  
The Open File dialog is displayed.  
Navigate to the IgorXOPs7\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.
5. In the Solution Explorer window, select the XOP1 icon.  
Choose Project→Add Existing Item.  
The Open File dialog is displayed.  
Navigate to the IgorXOPs7\XOPSupport\VC 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.
6. In the Solution Explorer window, select the IGOR64.lib and XOPSupport64.lib icons.  
Make sure that only IGOR64.lib and XOPSupport64.lib are selected.  
Right-click one of the selected icons and choose Properties.  
The Properties window opens.  
Select All Configurations from the Configuration menu.  
Select Win32 from the Platform menu.  
In the General category, set Excluded From Build to Yes.  
Click the Apply button.  
Click OK.
7. In the Solution Explorer window, select the IGOR.lib and XOPSupport.lib icons.  
Make sure that only IGOR.lib and XOPSupport.lib are selected.  
Right-click one of the selected icons and choose Properties.  
The Properties window opens.  
Select All Configurations from the Configuration menu.  
Select x64 from the Platform menu.  
In the General category, set Excluded From Build to Yes.  
Click the Apply button.  
Click OK.
8. Choose File→Save All.

The project is now configured for 64-bit as well as 32-bit compilation.

### Updating the 64-bit Visual C++ Target

1. Choose View→Solution Explorer.
2. Right-click the XOP1 icon and choose Properties to display the XOP1 Property Pages dialog.  
From the Configuration popup menu, choose All Configurations.  
From the Platform popup menu, choose x64.
3. Under Configuration Properties, select General and enter the following settings:  
*Output Directory:* .\  
*Target Name:* XOP1-64

## Appendix B — Project Changes For XOP Toolkit 7

---

*Target Extension:* .xop

*Character Set:* Use Multi-Byte Character Set

Click Apply.

4. Under Configuration Properties, select Debugging and enter the following settings:

*Command:* <path to your Igor64.exe application>

Click Apply.

5. Under C/C++, select General and enter the following settings:

*Additional Include Directories:* ..\..\XOPSupport;%(AdditionalIncludeDirectories)

*Debug Information Format:* Program Database (/Zi)

Click Apply.

6. Under C/C++, select Command Line and add the following setting:

*Additonal Options:* /utf-8

7. Under C/C++, select Preprocessor.

From the Configuration popup menu, choose Debug.

Verify that `_CRT_SECURE_NO_WARNINGS` is included as a preprocessor definition. If not, add it.

Click Apply.

From the Configuration popup menu, choose Release.

Verify that `_CRT_SECURE_NO_WARNINGS` is included as a preprocessor definition. If not, add it.

Click Apply.

8. Under C/C++, select Code Generation.

From the Configuration popup menu, choose Debug and enter the following settings:

*Runtime Library:* Multi-Threaded Debug (/MTd)

From the Configuration popup menu, choose Release and enter the following settings:

*Runtime Library:* Multi-Threaded (/MT)

From the Configuration popup menu, choose All Configurations and enter the following settings:

*Struct Member Alignment:* Default

Click Apply.

9. Under Linker, select General and enter the following settings:

*Output File:* XOP1-64.xop

Click Apply.

10. Under Linker, select Input and enter the following settings:

*Additional Dependencies:* version.lib;%(AdditionalDependencies)

Click Apply.

11. Under Linker, select System and enter the following settings:

*Enable Large Addresses:* Yes (/LARGEADDRESSAWARE)

Click Apply.

12. Under Linker, select Advanced and enter the following settings:

*Target Machine:* MachineX64

Click Apply.

13. Under Resources, select General and enter the following settings:

*Additional Include Directories:* ..\..\XOPSupport;%(AdditionalIncludeDirectories)

Click Apply.

14. Click OK to close the Property Pages dialog.



15. In the Solution Explorer, select the XOPSupport64.lib icon.  
Choose View→Properties Window.  
In the Properties window, set Relative Path to: ..\..\XOPSupport\VC\XOPSupport64.lib  
Press Enter to enter the setting.  
Close the Properties window.
16. Choose File→Save All.
17. From the Solution Platforms popup menu in the toolbar, choose x64.  
Choose Build→Build Solution.  
The project should successfully compile and link, producing XOP1-64.xop. You may get some warnings.

## Text Encoding Issues

In XOP Toolkit 7, we use UTF-8 text encoding for all C and C++ source files. This matters only if your XOP uses non-ASCII literal strings. See **Text Encodings** on page 201 for details.



# XOP Toolkit 7 Release History

## Overview

This appendix describes changes in each new release of XOP Toolkit 7.

For a discussion of what is new in XOP Toolkit 7 relative to XOP Toolkit 6, see **XOP Toolkit 7 Upgrade Notes** on page 415.

If you are targeting Igor Pro 8.00 or later, and you are creating a new XOP or updating an XOP already updated using XOP Toolkit 7.01 or later, you should use XOP Toolkit 8. See **Choosing Which XOP Toolkit to Use** on page 5 for details.

## Release 7.00

**July 25, 2016**

First release of XOP Toolkit 7.

## Release 7.01

**May 5, 2017**

XOP Toolkit 7.01 has been tested with Xcode 7, Xcode 8, Visual C++ 2015, and Visual C++ 2017.

Added WM memory XOPSupport routines to provide compatibility with Igor Pro 8 (in development as of this writing) and later. See **WM Memory XOPSupport Routines** on page 195 for details.

On Windows, in XOP Toolkit 7.00, the XOPStandardHeaders.h file included this:

```
#pragma execution_character_set("utf-8")
```

This tells the Visual C++ compiler to compile literal strings as UTF-8, as needed for Igor7 or later.

In Visual Studio 2015 Update 2, this pragma is deprecated in favor of using the new /utf-8 project setting and was removed from XOPStandardHeaders.h. If you are compiling with Visual C++, make sure that you are using Visual Studio 2015 Update 2 or later, and add the /utf-8 setting to your project. To do that, add the /utf-8 flag in the Command Line section of the C/C++ section of the property pages for all configurations of the project. For background information, see **Text Encodings in Visual C++** on page 204.

Fixed a bug in the ThreadGroupGetDF XOPSupport routine.

Fixed section **Converting Your Older Xcode Project for Xcode 9** on page 427 to include changing Exports.exp file in the Xcode folder from \_main to \_XOPMain.

Changed the Visual C++ “Program Database File Name” setting in the XOPSupport project to “\$(Out-Dir)\$(TargetName).pdb”. This gives each version of the XOPSupport library a unique .pdb file name and

## Appendix C — XOP Toolkit 7 Release History

---

puts it in the same folder as the library itself. XOP Toolkit 7.01 includes these XOPSupport.pdb files in order to eliminate LNK4099 warnings when compiling an XOP.

Changed the Visual C++ “Linker/Debugging/Generate Program Database File” setting in all projects from “\$(IntDir)\$(TargetName).pdb” to “\$(OutDir)\$(TargetName).pdb”. This is consistent with the value of that setting in new VC 2015 projects.

For clarity, changed the version field of the IOREC structure to XOPProtocolVersion and changed system field in XOPStuff structure to developmentSystem. These changes should have no impact on existing XOPs.

Changed the symbol MAX\_DIM\_LABEL\_CHARS, defined in IgorXOP.h, to MAX\_DIM\_LABEL\_BYTES.

Changed all sprintf calls in XOPSupport to snprintf.

The XOPI resources of all sample XOPs were changed to use the new XOP\_FEATURE\_FLAGS and XOPI\_RESERVED macros. It is recommended but not required that you update your XOP resource files.

On Macintosh, the new XOPI resource definition is:

```
resource 'XOPI' (1100) {
    XOP_VERSION,                // XOP protocol version.
    DEV_SYS_CODE,               // Code for development system used to make XOP
    XOP_FEATURE_FLAGS,          // Tells Igor about XOP features
    XOPI_RESERVED,              // Reserved - must be zero.
    XOP_TOOLKIT_VERSION,        // XOP Toolkit version.
};
```

On Windows, the new XOPI resource definition is:

```
1100 XOPI                      // XOPI - Describes general XOP properties to IGOR.
BEGIN
    XOP_VERSION,                // Version number of host XOP system.
    DEV_SYS_CODE,               // Code for development system used to make XOP.
    XOP_FEATURE_FLAGS,          // Tells Igor about XOP features
    XOPI_RESERVED,              // Reserved - must be zero.
    XOP_TOOLKIT_VERSION        // XOP Toolkit version.
END
```

Removed the PrepareLoadIgorData, LoadIgorData, EndLoadIgorData, and SaveIgorData callbacks. These callbacks were used only by the Data Browser XOP with Igor6 and earlier. As of Igor7, the Data Browser is built into Igor. Data structures used with these callbacks have changed and are no longer compatible with the old Data Browser XOP.

### 64-bit Integer Support

Added additional support for 64-bit integers. Before using 64-bit integer waves, read **64-bit Integer Issues** on page 215.

The following XOPSupport routines now support signed 64-bit integers (NT\_I64) and unsigned 64-bit integers (NT\_I64 | NT\_UNSIGNED):

**ScaleData, ScaleClipAndRoundData, FetchNumericValue, StoreNumericValue, MDGetTextWave-PointValue, MDSetTextWavePointValue, MDGetDPDataFromNumericWave, MDStoreDPDataInNumericWave**

Added new XOPSupport routines **MDGetNumericWavePointValueSInt64, MDSetNumericWavePointValueSInt64, MDGetNumericWavePointValueUInt64, and MDSetNumericWavePointValueUInt64.**

### Changes to NamedFIFO.h

The following changes were made to the NamedFIFO.h file. These changes affect programmers using FIFOs from XOPs only. They do not affect runtime behavior. They may cause a compile error when you recompile your XOP if it uses Igor FIFOs.

Changed the name field in the NamedFIFO structure to fifoName. Changed the size of this field from MAX\_OBJ\_NAME to MAX\_FIFO\_NAME but it is 31 in both cases.

Changed the name field in the ChartChanInfo structure to fifoChannelName. Changed the size of this field from MAX\_OBJ\_NAME to MAX\_FIFO\_CHANNEL\_NAME but it is 31 in both cases.

## Release 7.02

### June 4, 2018

XOP Toolkit 7.02 has been tested with Xcode 7 through Xcode 9, Visual C++ 2015, and Visual C++ 2017.

Added an explanation of when to use XOP Toolkit 8 versus XOP Toolkit 7. *If you are targeting Igor Pro 8.00 or later, and you are creating a new XOP or updating an XOP already updated using XOP Toolkit 7.01 or later, you should use XOP Toolkit 8.* See **Choosing Which XOP Toolkit to Use** on page 5 for details.

Added an explanation of how Igor Pro 8.00 and later handle long object names when calling an XOP compiled with XOP Toolkit 7 or before. See **Long Names and Paths in XOPs** on page 206 for details.

Fixed an asymptomatic bug in GBLoadWaveX, SimpleLoadWave and VDT by changing this:

```
static int fileFilterIndex = 0;
```

to this:

```
static int fileFilterIndex = 1;
```

This is appropriate because the XOPOpenFileDialog2 and XOPSaveFileDialog2 fileFilterIndex parameters are 1-based.

Fixed a bug in Igor that caused the ConvertTextEncoding callback to return an error for any error mode other than kTECNoMapErrorModeFail.

Fixed a crash in XOPEmergencyAlert when called from other than the main thread. That occurs if you call a non-threadsafe XOPSupport routine from a preemptive thread.

**GetPathInfo2** is thread-safe only when running with Igor Pro 8.00 or later. That means that, in Igor Pro 8.00 or later, you can use /P=<path name> when calling an XOP file loader from an Igor preemptive thread. You need to recompile your XOP with XOP Toolkit 7.02 or later to use this feature.

rom MAX\_OBJ\_NAME to MAX\_FIFO\_CHANNEL\_NAME but it is 31 in both cases.

## Release 7.03

### September 28, 2018

XOP Toolkit 7.03 adds Xcode 10 compatibility. Xcode 10 supports 64-bit development only and requires MacOS 10.13.6 or later.

For instructions on updating an XOP Toolkit 6 XOP for Xcode 10, see **Xcode Project Changes For XOP Toolkit 7** on page 427.

For instructions on updating an XOP Toolkit 7 XOP for Xcode 10, see **Updating Xcode Projects for Xcode 10** on page 44.



## Symbols

- #defines
  - XOP Toolkit 7 418
- Wno-parentheses 43
- \_CRT\_SECURE\_NO\_WARNINGS 50
- .c files 10
- .exp files
  - Xcode 36, 42
- .ihf extension 219
- .r files 10
  - compiled by Rez 56
- .rc files 10, 56
  - Visual C++ 48
- .sln files 48
- .vcxproj files 48
- .xop extension 13, 53

## Numbers

- 64-bit integers 215
- 64-bit XOPs 187
  - command help 193
  - data types 188–189
  - long data type 188, 190
  - printf and sprintf issues 192
  - programming issues 189
  - virtual memory 187
  - Windows 187

## A

- aborting 214
  - CheckAbort 392
- accessing data folders
  - XOPSupport routines 300–315
- accessing variables
  - XOPSupport routines 291–299
- accessing waves
  - XOPSupport routines 253–290
- activating XOPs 6
- Active Scheme
  - Xcode 39, 430–431
- Activity Monitor
  - leaks 227
- ActualToResourceItem 152, 317
- ActualToResourceMenuID 151, 316
- adding commands 71–98
- adding dialogs 207
- adding functions 99–119
- adding menus and menu items 145–153
- adding operations 71–98
- alerts
  - custom error alert 64

- XOPOKAlert 406
- XOPOKCancelAlert 406
- XOPYesNoAlert 407
- XOPYesNoCancelAlert 407
- aliases 13
- ALLOW\_MAC\_OS\_NATIVE\_MEMORY\_ROUTINES 200
- allWinOp 73
- architectures
  - Xcode 42
- ArrowCursor 419
- as keyword 76
  - optional parameters 78
- ASCII 201
- Asian text
  - ConcatenatePaths 330
  - FullPathPointsToFile 332
  - FullPathPointsToFolder 332
  - GetDirectoryAndFileNameFromFullPath 331
  - GetNativePath 330
  - HFSToPosixPath 329
  - MacToWinPath 328
  - strchr2 400
  - strrchr2 400
  - WinToMacPath 329
- automatic WAVE references 87
  - GetOperationDestWave 251
  - SetOperationWaveRef 247

## B

- background processing
  - SpinProcess 393
- backslashes
  - EscapeSpecialCharacters 401
  - UnEscapeSpecialCharacters 402
- balloon help 219
- base SDK
  - in Xcode 41
- BCInt data type 189
- big-endian
  - MDChangeWave2 268
- bugs 221–232
- build location
  - Xcode 39
- bundles
  - Xcode 37
- byte order
  - MDChangeWave2 268

## C

- C strings 110, 138

- C++
  - called from C code 217
  - calling C code 216
  - catch 217
  - exceptions 217
  - templates 127
  - try 217
  - writing XOPs in 216
- CalcWaveRange 85, 264
- callbacks 11, 13
  - IGOR.lib 11
  - not allowed from private threads 179
  - threads 178
- CallFunction 212, 375
  - from an Igor thread 213
  - from an independent module 213
  - ProcGlobal module 213
- cancelling
  - CheckAbort 392
- Can't load the executable error 43
- Can't run Intel XOP error 43
- Carbon
  - history 3
  - paths 205
  - XOP Toolkit 6 4
- catch
  - in C++ 217
- categories
  - for functions 102
  - of operations 73
- CatPossiblyQuotedName 362
- CFBundleExecutable 234
- CFM 3
- ChangeWave 254
  - may move wave in memory 127
- chart displays
  - XOPSupport routines 353
- CheckAbort 214, 392
- CheckFunctionForm 212, 374
- CheckName 360
- CheckRunningInMainThread 180, 379
- child data folder 305–306
- CHUNKS 128, 253, 281
- CLEANUP message 58
  - data folder reference counting 144
  - wave reference counting 134
- CleanupName 359
- CLEAR\_MODIFIED message 60, 66
- clipboard
  - XOPGetClipboardData 408
- CMD message 72
  - error codes 61
- CmpStr 400
- Cocoa 155, 165
- colons 205
- color tables
  - GetIgorColorTableInfo 364
  - GetIgorColorTableValues 365
  - GetIndexedIgorColorTableName 364
  - GetNamedIgorColorTableHandle 364
  - XOPSupport routines 364–365
- column-major order 128
- COLUMNS 128, 253, 281
- command help
  - 64-bit XOPs 193
- command template 75
- command templates 76
  - length limit 95
- command window
  - HistoryDelete 366
  - HistoryDisplaySelection 366
  - HistoryFetchParagraphText 367
  - HistoryFetchText 368
  - HistoryGetSelLocs 367
  - HistoryInsert 366
  - HistoryLines 366
  - HistorySetSelLocs 367
  - XOPSupport routines 366–369
- commands
  - adding 71–98
  - length limits 206
  - PauseUpdate 386
  - PutCmdLine 394
  - ResumeUpdate 386
  - XOPCommand 384
  - XOPCommand2 384
  - XOPCommand3 385
  - XOPSilentCommand 385
- commas
  - in operations 71
- communicating with Igor
  - XOPSupport routines 239–241
- compatibility 69–70
  - external operations 98
  - Igor 1.2 70
  - Igor Pro 70
  - of XOPs with Igor versions 4
  - operations 72
  - XOPI resource 57
- compilableOp 73
- compilableOp operation category 73
- compiler version
  - LLVM 4
  - Xcode 41
- compiling
  - help files 219
- complex conjugate
  - example external function 99
- complex conjugate example 108
- complex numbers
  - external function example 99



- parameters in external functions 108
- results in external functions 108
- storage in waves 128
- compound names 84
- computation threads 176
- ConcatenatePaths 330
- configurations
  - Visual C++ 49
- ConvertData 354
- ConvertData2 355
- ConvertTextEncoding 402
- CountInt data type 189
- CR\_STR macro 421
- Create New XOP Project 24
- CreateValidDataObjectName 360
- creating a project 24
- cross-platform development 201
  - files 204
- current data folder 303
- cursorOp 73
- curve fitting
  - function parameters 105
  - SimpleFit sample XOP 12, 20
  - SimpleGaussFit sample XOP 20
  - WAVE\_TYPE 105
- custom errors 63–64

## D

- dangling pointers 198, 228–229
  - waves 127
- data acquisition
  - DoXOPIdle operation 184
  - from threads 180–186
  - IDLE message 184
  - mutexes 184
  - WaveHandleModified 185
- data folder reference counting
  - CLEANUP message 144
  - NEW message 144
  - structure parameters 144
- data folder reference parameters
  - external operations 86
- data folder reference waves 135
- data folder references
  - external function example 107
  - in structure parameters 209
  - parameters in external functions 106
  - results in external functions 107
- data folders 141–144
  - accessing
    - XOPSupport routines 300–315
  - child 305–306
  - commonly used XOPSupport routines 142
  - conventions 143
  - current 303

- DataFolderAndName example 89
- DuplicateDataFolder 307
- DuplicateDataFolderObject 313
- FetchWaveFromDataFolder 255
- free 305
- GetCurrentDataFolder 303
- GetDataFolderByIDNumber 304
- GetDataFolderIDNumber 300
- GetDataFolderListing 302
- GetDataFolderNameOrPath 300
- GetDataFolderObject 310
- GetDataFolderProperties 301
- GetIndexedChildDataFolder 306
- GetIndexedDataFolderObject 309
- GetNamedDataFolder 303
- GetNumChildDataFolders 305
- GetNumDataFoldersObjects 308
- GetParentDataFolder 305
- GetRootDataFolder 303
- GetWavesDataFolder 306
- HoldDataFolder 315
- ID numbers 300, 304
- KillDataFolder 307
- KillDataFolderObject 312
- listing 302
- MoveDataFolder 308
- MoveDataFolderObject 314
- NewDataFolder 306
- Packages data folder 143
- parent 305
- reference counting 143
- ReleaseDataFolder 315
- RenameDataFolder 308
- RenameDataFolderObject 314
- root 303
- SetCurrentDataFolder 303
- SetDataFolderObject 311
- SetDataFolderProperties 301
- ThreadGroupGetDF 380
- ThreadGroupPutDF 380
- threads 177
- variables 291
- waves 306
- data full-scale 258–259
  - MDGetWaveScaling 269
  - MDSaveWaveScaling 269
- data sharing 69
- data types
  - 64-bit XOPs 188
  - BCInt 189
  - CountInt 189
  - IndexInt 189
  - PSInt 189
  - SInt16 188
  - SInt32 188
  - SInt64 188

- SInt8 188
- TickCountInt 188
- UInt16 188
- UInt32 188
- UInt64 188
- UInt8 188
- XOPIORecParam 192
- XOPIORecResult 191
- DATAFOLDER\_OBJECT
  - CheckName 360
  - CreateValidDataObjectName 360
- DATAFOLDER\_TYPE 124
  - data folder reference parameters 106
  - in XOPF resource 101
  - result in external functions 107
- dataFolderAndName keyword
  - Operation Handler 77
- DataFolderAndName parameters
  - example 87, 89
  - external operations 86
  - WAVE references 87
- DataFolderHandles 141
- dataFolderRef keyword
  - Operation Handler 77
- DataObjectValuePtr
  - GetDataFolderObject 310
  - SetDataFolderObject 311
- dataOp 73
- dates 405
- DateToIgorDateInSeconds 405
- debug configuration
  - Visual C++ 49
- debugging 232–233
  - in Visual C++ 51
  - symbolic debugging 232
  - tricks 233
  - Xcode 46
  - XOPNotice 232
- default help file name 220
- deleting waves
  - versus killing 131
- deployment target
  - in Xcode 41
- dereferencing a handle 229
- dereferencing a wave handle 230
- dereferencing handles 197
- destination waves 86–87
  - SetOperationWaveRef 247
- DEV\_SYS\_CODE 56–57
- development system 57
  - supported systems 35
- development systems
  - supported systems 6, 415
- DFREFs 86
  - data folder reference counting 144
  - in structure parameters 209
- dialogs
  - adding 207
  - DialogUtilities files 207
  - IgorError 393
  - XOPDisplayHelpTopic 395
  - XOPOpenFileDialog 339
  - XOPOpenFileDialog2 341
  - XOPSaveFileDialog 342
  - XOPSaveFileDialog2 344
  - XOPSupport routines 327
- DialogUtilities files 207
- dimension labels
  - GetWaveDimensionLabels 272
  - MDGetDimensionLabel 271
  - MDSetDimensionLabel 272
  - name length limits 206
  - SetWaveDimensionLabels 273
- dimension scaling
  - for waves 129
  - MDGetWaveScaling 269
  - MDSetWaveScaling 269
- dimensions
  - MDChangeWave 267
  - MDChangeWave2 268
  - MDGetWaveDimensions 266
  - MDMakeWave 265
  - ROWS, COLUMNS, LAYERS, CHUNKS 128, 253
- direct access method 126
- direct functions 116
  - resources 116, 118
- direct method for external functions 116
- direct method for external operations 72
- direct operations
  - resources 97
- disabling menu items 152–153
- displayOp 73
- DisposeHandle 199–200
- DisposePtr 200
- DLLs
  - using in XOPs 217
- DoFunction routine 116
- double name parameters
  - external operations 84
  - independent modules 84
- doubleName keyword
  - Operation Handler 77
- DoUpdate 386
  - recursion 68
- DoWindowRecreationDialog 372
- DoXOPIdle operation 68, 184
  - data acquisition 184
- drawOp 73
- DuplicateDataFolder 307
- DuplicateDataFolderObject 313

## E

- email 18
- enabling menu items 152–153
- endian
  - MDChangeWave2 268
- EndLoadIgorData 442
- error codes 61
  - from operations 72
- errors 61–64
  - custom error alert 64
  - custom errors 63–64
  - custom XOP error codes 62
  - error codes 61
  - external functions 104
  - FIRST\_XOP\_ERR 64
  - GetIgorErrorMessage 61, 64, 393
  - Igor error codes 61
  - IgorError 61, 64, 393
  - in external functions 118
  - Mac OS error codes 62
  - STR# 1100 resource 63
  - Windows OS error codes 62
  - WindowsErrorToIgorError 382–383
  - XOPOKAlert 64
  - XOPOKCancelAlert 64
  - XOPYesNoAlert 64
  - XOPYesNoCancelAlert 64
- escape sequences
  - EscapeSpecialCharacters 401
  - UnEscapeSpecialCharacters 402
- EscapeSpecialCharacters 401
- exceptions
  - in C++ 217
- ExecuteOperation 74
  - extern "C" declaration 94
  - NULL handles 95
- EXP\_xxx 60–61
- experiment settings 59–61
- experiment type argument 60–61
- experiments 65–66
  - CLEAR\_MODIFIED message 60
  - EXP\_xxx 60–61
  - LOAD message 60
  - LOAD\_TYPE\_xxx 61
  - loading settings 65
  - LOADSETTINGS message 60
  - MODIFIED message 60
  - NEW message 59
  - SAVE message 60
  - SAVE\_TYPE\_xxx 60
  - SAVESETTINGS message 60
  - saving settings 65
- extended structure parameters 92
  - example in external operation 93
- extensions
  - help files 219
  - help files for 64-bit XOPs 193
- extern "C" declaration
  - calling C from C++ 216
  - ExecuteOperation 94
  - external functions 103
  - in C++ called from C 217
  - with routines called by Igor 55
- external functions 99–119
  - adding 100
  - calling from an XOP 212
  - complex parameters 108
  - complex results 108
  - data folder reference parameters 106
  - data folder reference results 107
  - direct functions 116
  - direct method 116
  - DoFunction routine 116
  - error codes 104, 118
  - examples 99
  - extern "C" declaration 103
  - FUNCADDRS message 59, 116
  - function categories 102
  - FUNCTION message 116
  - FV\_REF\_TYPE 114
  - guided tour 20
  - invoking from Igor 103
  - keep RESIDENT 116
  - logfit example 118
  - message method 116
  - names of 100, 102
  - optional parameters 99
  - parameter types 100, 104
  - parameters 99, 103
  - pass-by-reference 114
  - pass-by-value 114
  - resources 116, 118
  - result types 100, 104
  - results 104
  - returning results 103
  - SimpleFit sample XOP 12
  - string parameters 108–109, 111, 115
  - string results 108–109
  - structure alignment 103
  - structure parameter example 112
  - structure parameters 111–112
  - thread-safe 117
  - tp pointer 117
  - versus user functions 99
  - wave assignment statements 103
  - wave parameters 105, 119
  - wave results 106
  - WaveAccess sample XOP 12
  - XFUNC1 sample XOP 11
  - XFUNC2 sample XOP 11

XFUNC3 sample XOP 11  
 external operations 71–98  
   as keyword 76  
   command template 75  
   command templates 76  
   compatibility 98  
   compatibility with Igor Pro 4 72  
   compilableOp 73  
   compilableOp operation category 73  
   data folder reference parameters 86  
   DataFolderAndName parameters 86  
   double name parameters 84  
   extended structure parameter example 93  
   extended structure parameters 92  
   flags 71, 75  
   GBLoadWaveX sample XOP 12  
   GetOperationWaveRef 248  
   keywords 75  
   kOperationIsThreadSafe 96  
   MenuXOP1 sample XOP 11  
   mnemonic names 76  
   name parameters 84  
   names of 74  
   NIGPIB2 sample XOP 12  
   Operation Handler 74–98  
   Operation Handler XOPSupport routines 242  
   optional parameters 77, 81  
   output variables 94  
   parameters 71, 75  
   RegisterOperation 242  
   resources 97  
   runtime parameter structure 79  
   SimpleLoadWave sample XOP 12  
   starter code 74  
   string parameters 83  
   structure parameter example 90  
   structure parameters 89  
   template 75  
   templates 76  
   thread-safe 95  
   tp pointer 96  
   variables 242–246, 347  
   VarName parameters 85  
   VDT2 sample XOP 12  
   vs keyword 76  
   wave parameters 84  
   wave references 247–248  
   WaveRange parameters 84  
   XOP1 sample XOP 11  
   XOPOp operation category 73

## F

F\_ANLYZWAVES 102  
 F\_CMPLX 102  
 F\_CONV 102

F\_EXP 102  
 F\_EXTERNAL 102  
 F\_EXTERNAL function category 102  
 F\_IO 102  
 F\_NUMB 102  
 F\_ROUND 102  
 F\_SPEC 102  
 F\_STR 102  
 F\_THREADSAFE 102, 117  
 F\_TIMEDATE 102  
 F\_TRIG 102  
 F\_UTIL 102  
 F\_WAVE 102  
 F\_WINDOWS 102  
 FetchNumericDataUsingVarName 86, 244  
 FetchNumericValue 283  
 FetchNumVar 292  
   example 140  
 FetchStrHandle 294  
 FetchStringDataUsingVarName 86, 246  
 FetchStrVar 293  
   example 140  
 FetchWave 255  
 FetchWaveFromDataFolder 255  
 fields  
   of structure parameters 209  
 FIFOs  
   GetNamedFIFO 353  
   MarkFIFOUpdated 353  
   XOPSupport routines 353  
 file loaders  
   FileLoaderMakeWave 346  
   GBLoadWave sample XOP 12  
   GetFullPathFromSymbolicPathAndFilePath 345  
   Load Waves submenu 12  
   output variables 94  
   SanitizeWaveName 359  
   SetFileLoaderOperationOutputVariables 94, 347  
   SetFileLoaderOutputVariables 94, 346  
   SimpleLoadWave sample XOP 12  
   XOPOpenFileDialog 339  
   XOPOpenFileDialog2 341  
   XOPSaveFileDialog 342  
   XOPSaveFileDialog2 344  
   XOPSupport routines 345–348  
 FILE\_LOADER flags 346  
 FileLoaderMakeWave 346  
 filenames  
   illegal characters 328–329  
 files  
   accessing  
     XOPSupport routines 328–344  
   ConcatenatePaths 330  
   cross-platform routines 204

- FullPathPointsToFile 332
- FullPathPointsToFolder 332
- GetDirectoryAndFileNameFromFullPath 331
- GetLeafName 332
- GetNativePath 330
- HFSToPosixPath 329
- MacToWinPath 328
- MAX\_DIRNAME\_LEN 205
- MAX\_FILENAME\_LEN 205
- MAX\_PATH\_LEN 205
- MAX\_VOLUMENAME\_LEN 205
- ParseFilePath 333
- path conversions 205
- path length limits 206
- path separator characters 205
- platform independence 12
- SpecialDirPath 333
- WinToMacPath 329
- XOPAtEndOfFile 338
- XOPCloseFile 335
- XOPCreateFile 334
- XOPDeleteFile 334
- XOPGetFilePosition 337
- XOPGetFilePosition2 338
- XOPNumberOfBytesInFile 339
- XOPNumberOfBytesInFile2 339
- XOPOpenFile 334
- XOPReadFile 335
- XOPReadFile2 335
- XOPReadFile64 336
- XOPReadLine 336
- XOPSetFilePosition 338
- XOPSetFilePosition2 338
- XOPWriteFile 337
- XOPWriteFile64 337
- find
  - menu item 317
- FIRST\_XOP\_ERR 62, 64
- FIRSTCMD 394
- FIRSTCMDCRHIT 394
- FixByteOrder 357
- flags 71, 75
  - optional parameters 77
  - prefix characters 76
- forward slashes 205
- frameworks
  - Xcode 37
- free data folders
  - GetParentDataFolder 305
- free waves 135
  - GetOperationDestWave 250
  - HoldWave 135
  - KillWave 135
  - MDMakeWave 266
  - reference counting 135
  - reference counting examples 135

- ReleaseWave 135
- FullPathPointsToFile 332
- FullPathPointsToFolder 332
- FUNCADDRS message 59, 116
- FUNCREFs
  - CallFunction from an independent module 213
  - in structure parameters 209
  - using from an XOP 212
- function categories 102
  - F\_ANLYZWAVES 102
  - F\_CMPLX 102
  - F\_CONV 102
  - F\_EXP 102
  - F\_EXTERNAL 102
  - F\_IO 102
  - F\_NUMB 102
  - F\_ROUND 102
  - F\_SPEC 102
  - F\_STR 102
  - F\_THREADSAFE 102
  - F\_TIMEDATE 102
  - F\_TRIG 102
  - F\_UTIL 102
  - F\_WAVE 102
  - F\_WINDOWS 102
- function index 116
- FUNCTION message 116
  - error codes 61
- functions
  - adding 99–119
  - CallFunction 212
  - calling from an XOP 212
  - CheckFunctionForm 212, 374
  - GetFunctionInfo 212, 372, 375
  - GetFunctionInfoFromFuncRef 212, 373
  - GetIgorCallerInfo 377
  - GetIgorRTStackInfo 377
- FV\_REF\_TYPE 114
  - in XOPF resource 101
- FV\_STRUCT\_TYPE 112
  - in XOPF resource 101–102

## G

- GBLoadWave
  - relation to GBLoadWaveX 417
- GBLoadWave sample XOP 12
- GBLoadWaveX
  - relation to GBLoadWave 417
- GetCStringFromHandle 129, 138, 399
- GetCurrentDataFolder 303
- GetDataFolderByIDNumber 304
- GetDataFolderIDNumber 300
- GetDataFolderListing 302
- GetDataFolderNameOrPath 300
- GetDataFolderObject 310

GetDataFolderProperties 301  
GetDirectoryAndFileNameFromFullPath 331  
GetFullPathFromSymbolicPathAndFilePath 345  
GetFunctionInfo 212, 372  
GetFunctionInfoFromFuncRef 212, 373  
GetHandleSize 199–200  
GetIgorCallerInfo 377  
GetIgorColorTableInfo 364  
GetIgorColorTableValues 365  
GetIgorErrorMessage 61, 64, 393  
GetIgorProcedure 370  
GetIgorProcedureList 370  
GetIgorRTStackInfo 377  
GetIndexedChildDataFolder 306  
GetIndexedDataFolderObject 309  
GetIndexedIgorColorTableName 364  
GetLeafName 332  
GetMenuHandle 151  
GetNamedDataFolder 303  
    example 143  
GetNamedFIFO 353  
GetNamedIgorColorTableHandle 364  
GetNativePath 206, 330  
GetNumChildDataFolders 305  
GetNumDataFoldersObjects 308  
GetNVAR 210, 297  
    example 211  
GetOperationDestWave  
    example 87  
GetParentDataFolder 305  
GetPathInfo2 391  
GetPrefsState 65, 398  
GetPtrSize 200  
GetRootDataFolder 303  
    example 143  
GetSVAR 210, 298  
    example 211  
GetTextWaveData 129, 286  
GetWaveDimensionLabels 272  
GetWavesDataFolder 306  
GetXOPIndString 352  
GetXOPItem 55, 57, 241  
    items field in IORecHandle 67  
    pitfalls 233  
    replace long data type 192  
GetXOPMessage 55, 57, 240  
    message field in IORecHandle 67  
    pitfalls 233  
    replace long data type 192  
GetXOPNamedResource 351  
GetXOPPrefsHandle 65, 397  
GetXOPResource 351  
GetXOPResult 240  
GetXOPStatus 241  
global variables 221  
    threads 176

GPIO communications  
    NIGPIB2 sample XOP 12  
GRAF\_MASK 397  
graphOp 73

## H

HandAndHand 200  
HandCursor 419  
Handles  
    disposing 83  
    GetCStringFromHandle 129, 138  
    PutCStringInHandle 129, 138  
    WaveMemorySize 263  
handles 195, 197–199  
    containing strings 233  
    dangling pointers 229  
    dereferencing 197, 229  
    DisposeHandle 199–200  
    GetHandleSize 199–200  
    GetPtrSize 200  
    HandAndHand 200  
    HandToHand 200  
    locking 198  
    Macintosh 199  
    master pointers 197  
    NewHandle 199–200  
    ownership rules 196  
    recommended practices 231  
    resizing 198  
    SetHandleSize 199–200  
    treating as C strings 110, 138  
    usage 198  
    WMDiscardHandle 200  
    WMGetHandleSize 200  
    WMHandAndHand 200  
    WMHandToHand 200  
    WMNewHandle 200  
    WMSetHandleSize 200  
HandToHand 200  
hdlg resources 423  
header files 9  
    changes in XOP Toolkit 7 418  
    IgorErrors.h 418  
    XOPWindows.h 418  
heap 197  
    fragmentation 198  
    overwriting 222  
help 219–220  
    balloon help 219  
    command help 31  
    for 64-bit XOPs 193  
    for operations and functions 219–220  
    function categories 102  
    help file name 220  
    Igor Help Browser 219

- operation categories 73
- searching for help files 395
- STR# 1101 resource 220
- technical support 17
- tips 396
- topics and subtopics 395
- XOPDisplayHelpTopic 220, 395
- XOPSetContextualHelpMessage 396
- Help Browser 31
- help files
  - compiling 219
  - extensions 219
  - for 64-bit XOPs 193
  - location of 219
- HFS
  - paths 205, 328
- HFSToPosixPath 206, 329
- HGetState 198
- history
  - cistory carbon copy 388
  - XOPNotice 386
  - XOPNotice2 387
  - XOPNotice3 388
  - XOPResNotice 389
- history area
  - HistoryDelete 366
  - HistoryDisplaySelection 366
  - HistoryFetchParagraphText 367
  - HistoryFetchText 368
  - HistoryGetSelLocs 367
  - HistoryInsert 366
  - HistoryLines 366
  - HistorySetSelLocs 367
- HistoryDelete 366
- HistoryDisplaySelection 366
- HistoryFetchParagraphText 367
- HistoryFetchText 368
- HistoryGetSelLocs 367
- HistoryInsert 366
- HistoryLines 366
- HistorySetSelLocs 367
- HLock 198
- HMNU resources 423
- hmnu resources 423
- HMODULE 382
- HoldDataFolder 143, 315
- HoldWave 132, 289
  - free waves 135
  - wave reference waves 134
- HSetState 198
- HSTRING\_TYPE 109, 115
  - in XOPF resource 101
- HUnlock 198
- HWND
  - IgorClientHWND 382

## I

- IBeamCursor 419
- ID numbers for data folders 300, 304
- IDLE message 58
  - data acquisition 184
  - recursion 68–69
- IDLE messages
  - DoXOPIdle operation 68, 184
- IDLES bit 58, 67
  - SetXOPType 239
- idling 66
- IEEE\_FLOAT 354
- Igor 1.2
  - compatibility 70
- Igor commands 384–385, 394
- Igor Extensions folder 6, 13, 72, 100
- Igor Help Browser 219
- Igor mailing list 18
- Igor number type codes 354, 356
- Igor Pro
  - compatibility 70, 195
  - first Windows version 3
  - Igor Extensions folder 6
  - 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 6.20 195
  - Igor Pro 6.3 4
  - Igor User Files folder 6
- Igor Pro 4
  - external operations 72
- Igor Pro User Files
  - SpecialDirPath 333
- Igor Tips
  - XOPSetContextualHelpMessage 396
- Igor User Files folder 6
- Igor-thread-safe 178
- IGOR\_OBSOLETE 70, 421
- IGOR.lib
  - callback routines 11
  - handles 199
  - link with 233
  - memory management 199
  - XOPSupport 9
- IGOR32 187
- IGOR32 #define 418
- IGOR64 187
- IGOR64 #define 418
- IgorClientHWND 382
- IgorColorSpec 365
- IgorDateInSecondsToDate 405
- IgorError 61, 64, 393

- IgorErrors.h 418
- IgorExchange 18
- IgorModule 382
- IgorVersion 395
- igorVersion 70
- igorVersion global 70
- IgorXOP.h 9
- IgorXOPs 7 folder
  - copy existing XOPs to 415
- IgorXOPs6 folder
  - contents of 8
  - creating 7
- IGR0 XOP file creator 13, 53
  - Xcode 43
- illegal characters 328–329
- independent modules
  - CallFunction 213
  - double name parameters 84
- IndexInt data type 189
- Info.plist
  - Xcode 37, 42
- Info.plist file 43, 234
- Info64.plist 234
- InfoPlist.strings file 43
- INFs 404
- INIT message 13, 58
  - error codes 61
- initialization 13, 58, 239
- INSERTCMD 394
- installing XOP Toolkit 7
- integers
  - 64 bit versus doubles 215
- Intel 3
  - compiling in Xcode 25
  - initial support of 3
- internal threads 176
  - callbacks not allowed 179
  - data acquisition 184–185
- ioOp 73
- IORecHandle 13, 53, 55, 66–67
  - passed to main 17
  - recursion 68
- IsINF32 404
- IsINF64 404
- IsNaN32 404
- IsNaN64 404
- ITEM\_REQUIRES bits 152
- itemFlags field 152
- IXOP XOP file type 13, 53
  - Xcode 43

## K

- keywords 75
  - in operations 71
  - optional parameters 77

- prefix characters 76
- KillDataFolder 307
- KillDataFolderObject 312
- killing waves 130
  - preventing killing 130
  - versus deleting 131
- KillWave 255
  - free waves 135
- kMDWaveAccessMode0 127, 273
- kOpDestWaveChangeExistingWave 249
- kOpDestWaveMakeFreeWave 250
- kOpDestWaveMustAlreadyExist 250
- kOpDestWaveOverwriteExistingWave 250
- kOpDestWaveOverwriteOK 249
- kOperationIsThreadSafe 96
- kXOPWindowMessageUpdate
  - recursion 68
- kXOPWindowMessageUpdate message
  - recursion 68

## L

- LAYERS 128, 253, 281
- layoutOp 73
- leaks 227
- Legendre polynomials 11, 99
- liberal names 358
  - CatPossiblyQuoteName 362
  - CleanupName 359
  - CreateValidDataObjectName 360
  - PossiblyQuoteName 362
- libraries
  - link errors 233
  - XOPSupport 9
- libXOPSupport.a 37
- libXOPSupport64.a 37
- linking
  - errors 233
- little-endian
  - MDChangeWave2 268
- LLVM compiler 4
  - main function issue 195
- LOAD message 60
- Load Waves submenu 12
- LOAD\_TYPE\_xxx 61
- LoadIgorData 442
- loading settings 65
- LOADSETTINGS message 60, 65
- lock state
  - waves 258
- locking handles 198
- logfit
  - example external function 11, 99, 118
- long data type
  - 64-bit XOPs 188, 190
  - replacing 191–192



# M

- Mach-O 3
  - creating projects in Xcode 36
- MACIGOR #define 418
- Macintosh
  - paths 205
  - supported OS versions 4
  - XOP Toolkit 6 requirements 4
- Macintosh menu manager routines 145
- MacOS
  - first Igor version 3
- MacRectToWinRect 407
- MacRoman 201
- MacToWinPath 206, 328
- mailing list 18
- main 13
  - called by Igor 53
  - LLVM compatibility 195
  - receives INIT message 58
  - receives IORecHandle 17
  - version checking 70
  - XOPMain 195
- main function 13
  - changed in XOP Toolkit 6 421
- main thread 176
- MAIN\_NAME\_SPACE 358
- MakeWave 253
- MallocDebug
  - leaks 227
- MarkFIFOUpdated 353
- master pointers 197
- MAX\_DIM\_LABEL\_BYTES 5, 442
- MAX\_DIM\_LABEL\_CHARS 271–273, 442
- MAX\_DIMENSIONS 265, 267
- MAX\_DIRNAME\_LEN 205
- MAX\_FILENAME\_LEN 205
- MAX\_OBJ\_NAME 5
- MAX\_PATH\_LEN 5, 205
- MAX\_UNIT\_CHARS 259–260, 270–271
- MAX\_VOLUMENAME\_LEN 205
- MAX\_WAVE\_NAME 5
- MAXCMDLEN 5
- MDAccessNumericWaveData 126, 273
- MDChangeWave 267
  - may move wave in memory 127
- MDChangeWave2 268
  - may move wave in memory 127
- MDGetDimensionLabel 271
- MDGetDPDataFromNumericWave 125, 280
- MDGetNumericWavePointValue 125, 276
- MDGetNumericWavePointValueSInt64 278
- MDGetNumericWavePointValueUInt64 279
- MDGetTextWavePointValue 284
  - example 129
- MDGetWaveDimensions 128, 266
- MDGetWaveScaling 269
- MDGetWaveUnits 270
- MDI
  - getting Igor client HWND 382
- MDMakeWave 265
  - example 123
- MDSetDimensionLabel 272
- MDSetNumericWavePointValue 125, 276
- MDSetNumericWavePointValueSInt64 278
- MDSetNumericWavePointValueUInt64 280
- MDSetTextWavePointValue 285
  - example 129
- MDSetWaveScaling 129, 269
  - example 123
- MDSetWaveUnits 129, 271
- MDStoreDPDataInNumericWave 125, 281
- MemClear 399
- MemError 200
- memory
  - 64-bit XOPs 187
  - checking allocations 227
  - dangling pointers 228–229
  - data sharing 69
  - disposing 226–227
  - handles 197–199
  - heap 197
  - leaks 226–227
  - pointers 197
  - relocatable block 198
  - using unallocated blocks 225
- WMDisposeHandle 412
- WMDisposePtr 413
- WMGetHandleSize 410
- WMGetPtrSize 412
- WMHandAndHand 411
- WMHandToHand 411
- WMNewHandle 410
- WMNewPtr 412
- WMPtrAndHand 413
- WMPtrToHand 413
- WMSetHandleSize 410
- WMSetPtrSize 412
- memory management
  - Macintosh handles 199
- memory management routines 410–413
- menu handles 151
- menu IDs 150–151
- menu manager routines 145
- MENUENABLE message 58, 152
- MENUHELP resources 423
- MENUITEM message 58
  - error codes 61
  - recursion 68
- menus
  - ActualToResourceItem 152, 317
  - ActualToResourceMenuID 151, 316

- adding 145–153
- adding a main menu 148
- disabling 152–153
- enabling 152–153
- enabling Igor items 153
- Find item 317
- getting menu handles 151
- limitations 147
- menu IDs 150–151
- menu manager routines 145
- MENUENABLE message 58
- MENUIITEM message 58
- MenuXOP1 sample XOP 11, 145, 147
- resource IDs 150
- resources 147–150
- ResourceToActualItem 151, 316
- ResourceToActualMenuID 151, 316
- SetIgorMenuItem 317
- SHOW\_MENU\_AT\_LAUNCH 148
- SHOW\_MENU\_WHEN\_ACTIVE 148
- Undo item 317
- WM\_COMMAND message 150
- WM\_INITMENU message 152
- XMI1 1100 resource 147
- XMN1 1100 resource 147–148
- XOPActualMenuIDToMenuRef 318
- XOPAppendMenuItem 321
- XOPCheckMenuItem 324
- XOPCountMenuItems 319
- XOPDeleteMenuItem 322
- XOPDeleteMenuItemRange 322
- XOPDisableMenuItem 323
- XOPEnableMenuItem 323
- XOPFillMenu 317
- XOPFillMenuNoMeta 324
- XOPFillPathMenu 325
- XOPFillWaveMenu 325
- XOPFillWinMenu 326
- XOPGetMenuInfo 319
- XOPGetMenuItemInfo 320
- XOPGetMenuItemText 321
- XOPHideMainMenu 320
- XOPInsertMenuItem 322
- XOPMenuRef 147
- XOPResourceMenuIDToMenuRef 318
- XOPSetMenuItemText 321
- XOPShowMainMenu 319
- XOPSupport routines 316–326
- XSM1 1100 resource 147
- MenuXOP1 sample XOP 11
- MenuXOP1 XOP 145, 147
- message method for external functions 116
- messages
  - basic 57–59
  - between Igor and XOP 67
  - CLEANUP message 58

- CLEAR\_MODIFIED message 60
- from Igor to XOP 13
- FUNCADDRS message 59, 116
- IDLE message 58
- INIT message 13, 58
- LOAD message 60
- LOADSETTINGS message 60, 65
- MENUENABLE message 58
- MENUIITEM message 58
- MODIFIED message 60
- NEW message 59, 65
- OBJINUSE 130
- OBJINUSE message 59
- recursion 68
- SAVE message 60
- SAVESETTINGS message 60, 65
- saving and loading settings 59–61
- threads 176
- XOPEntry routine 13
- meta-characters 326
- XOPFillMenu 317
- XOPFillMenuNoMeta 324
- Microsoft 3
- miscellaneous XOPSupport routines 384–398
- mnemonic names 76
- modification count
  - waves 257
- modification date
  - waves 257
- modification state
  - waves 257
- MODIFIED message 60
- MoveDataFolder 308
- MoveDataFolderObject 314
- MoveLockHandle 198
- Multitasking
  - see threads
- Multithreading 175
  - see also threads
- mutexes 177
  - data acquisition 184
- MW\_MASK 397
- My Mac 32-bit 39
- My Mac 64-bit 39

## N

- name keyword
  - Operation Handler 77
- name parameters
  - external operations 84
- names
  - CatPossiblyQuotedName 362
  - CheckName 360
  - CleanupName 359
  - CreateValidDataObjectName 360

- liberal 359–360, 362
- object name length limits 206
- of external functions 102
- of external operations 74
- PossiblyQuoteName 362
- SanitizeWaveName 359
- standard 362
- UniqueName 358
- UniqueName2 358
- WaveName 261
- NaNs 404–405
- National Instruments
  - NIGPIB2 sample XOP 12
- native paths 206
- NEW message 59, 65
  - data folder reference counting 144
  - wave reference counting 134
- NewDataFolder 306
  - example 143
- NewHandle 199–200
- NewPtr 200
- NIGPIB2 sample XOP 12
- NT\_CMPLX 108
  - in XOPF resource 101
  - number type for waves 124
- NT\_FP32 124, 253, 354
  - number type for waves 124
- NT\_FP64 124, 253, 354
  - in XOPF resource 101
  - number type for waves 124
- NT\_I16 124, 253, 354
  - number type for waves 124
- NT\_I32 124, 253, 354
  - number type for waves 124
- NT\_I64 124
- NT\_I8 124, 253, 354
  - number type for waves 124
- NT\_UNSIGNED 124, 253, 354
  - number type for waves 124
- number keyword
  - Operation Handler 77
- number type codes 354, 356
- number types
  - for waves 124
  - GBLoadWave example 12
- NumBytesAndFormatToNumType 356
- numeric conversion
  - ConvertData 354
  - ConvertData2 355
  - FixByteOrder 357
  - NumBytesAndFormatToNumType 356
  - NumTypeToNumBytesAndFormat 356
  - ScaleClipAndRoundData 357
  - ScaleData 356
- numeric conversion routines 354–357
  - GBLoadWave example 12

- numeric expressions
  - in operations 71
- numeric precision
  - in external functions 104
- NumTypeToNumBytesAndFormat 356
- NVAR 85
- NVARs
  - GetNVAR 297
  - in structure parameters 209, 211
  - SetNVAR 298

## O

- object names
  - length limits 206
  - XOPSupport routines 358–363
- object types
  - WAVE\_OBJECT 59
- Objective-C++ 156, 165
- objects
  - counting in data folders 308
  - duplicating in data folders 313
  - getting in data folders 309–310
  - killing in data folders 312, 314
  - renaming in data folders 314
  - setting in data folders 311
- OBJINUSE message 59, 130
- OLD\_IGOR 421
- operation categories 73
  - allWinOp 73
  - compilableOp 73
  - cursorOp 73
  - dataOp 73
  - displayOp 73
  - drawOp 73
  - graphOp 73
  - ioOp 73
  - layoutOp 73
  - printOp 73
  - statsOp 73
  - tableOp 73
  - threadSafeOp 73
  - utilOp 73
  - waveOp 73
  - XOPOp 73
- Operation Handler 74–98
  - introduced in Igor Pro 5 3
  - RegisterOperation 242
  - XOPSupport routines 242
- operations
  - adding 71–98
- optional parameters 77, 81
  - as keyword 78
  - in external functions 99
  - vs keyword 78
- output variables 94

## P

## packages

Xcode 36

Packages data folder 143

PANEL\_MASK 397

## parameter types

in external functions 104

numeric precision 104

## parameters 75

as keyword 76

checking string parameters in external functions 111

commas between parameters 71

complex in external functions 108

data folder reference in external functions 106

data folder reference parameters 86

DataFolderAndName parameters 86

disposing string parameters in external functions 111

double name parameters 84

external functions 103

FV\_REF\_TYPE 114

in external functions 99–100

in operations 71

name parameters 84

optional 77

optional parameters 81

pass-by-reference 114

pass-by-value 114

runtime parameter structure 79

string in external functions 108–109, 115

string parameters 83

structures 209

structures in external functions 111–112

structures in external operations 89

threads 177

type checking in external functions 101

VarName parameters 85

vs keyword 76

wave in external functions 105, 119

wave parameters 84

WaveRange parameters 84

parent data folder 305

ParseFilePath 333

ParseOperationTemplate 74

## parsing commands

GetFullPathFromSymbolicPathAndFilePath  
345

pass-by-reference 114

FV\_REF\_TYPE 114

pass-by-value 114

path separator characters 205

PathList 390

## paths

Carbon 205

ConcatenatePaths 330

escape characters 205

file path length limits 206

FullPathPointsToFile 332

FullPathPointsToFolder 332

GetDirectoryAndFileNameFromFullPath 331

GetFullPathFromSymbolicPathAndFilePath  
345

GetLeafName 332

GetNativePath 206, 330

HFS 205, 328

HFSToPosixPath 329

illegal characters 328–329

Mac/Win conversion 205

Macintosh 205

MacToWinPath 206, 328

native paths 205

ParseFilePath 333

POSIX 205, 328

Posix 206

POSIX paths 205

separator 328

SpecialDirPath 333

WinToMacPath 206, 329

PauseUpdate 386

pitfalls 233–235

PL\_MASK 397

platform-independence 6

## plgndr

example external function 11, 99

point access method 125

## pointers 197

dangling pointers 228–229

misconceptions 224

POSIX 205

paths 205, 328

Posix 206

PossiblyQuoteName 362

## predefined macros

XOP Toolkit 7 418

## preferences 65

GetPrefsState 398

GetXOPPrefsHandle 397

SaveXOPPrefsHandle 397

SpecialDirPath 333

structures 65

prefix characters 76

## prefix files

Xcode 37, 42

PrepareLoadIgorData 442

## printf

64-bit XOPs 192

printOp 73

## private threads 176

callbacks not allowed 179

data acquisition 184–185

problems 17  
 procedures  
     CallFunction 212  
     calling from an XOP 212  
     CheckFunctionForm 212, 374  
     DoWindowRecreationDialog 372  
     GetFunctionInfo 212, 372, 375  
     GetFunctionInfoFromFuncRef 212, 373  
     GetIgorCallerInfo 377  
     GetIgorProcedure 370  
     GetIgorProcedureList 370  
     GetIgorRTStackInfo 377  
     SetIgorProcedure 371  
     XOPSupport routines 370–378  
 ProcGlobal  
     CallFunction 213  
 programming utilities  
     TickCount 405  
 programming utility routines 399–409  
 project files 10  
     Visual C++ 48  
 project settings  
     Visual C++ 49  
     Xcode 40  
 Project Settings dialog  
     Xcode 38  
 projects  
     Create New XOP Project 24  
     creating in guided tour 24  
     creating in Visual C++ 48  
     creating in Xcode 36  
     debug and release configurations 49  
 Property List Editor 234  
 PSInt data type 189  
 PtrAndHand 200  
 PtrToHand 200  
 PutCmdLine 394  
 PutCStringInHandle 129, 138, 399

## Q

qualified keywords 82  
 queues  
     thread output queue 182–183  
 quotes  
     CatPossiblyQuoteName 362  
     PossiblyQuoteName 362

## R

rectangles 407  
 recursion 67–69  
     DoUpdate side-effect 386  
     IDLE messages 68–69  
     XOPCommand side-effect 384  
     XOPCommand2 side-effect 384  
     XOPCommand3 side-effect 385

    XOPSilentCommand side-effect 385  
 reference counting  
     data folder reference waves 135  
     data folders 143  
     free wave examples 135  
     free waves 135  
     wave reference waves 134  
     waves 131  
 RegisterFunction  
     replace long data type 191  
 RegisterOperation 17, 72, 74, 242  
     example 53  
 release configuration  
     Visual C++ 49  
 ReleaseDataFolder 143, 315  
     data folder reference waves 135  
 ReleaseWave 132, 289  
     free waves 135  
     wave reference waves 134  
 relocatable block of memory 197–198  
 RenameDataFolder 308  
 RenameDataFolderObject 314  
 REPLACEALLCMDS 394  
 REPLACEALLCMDSCRHIT 394  
 REPLACEFIRSTCMD 394  
 RESIDENT bit 66  
     SetXOPType 239  
 resident XOPs  
     external functions 116  
     XOPType field in IORec 66  
 resource files 10  
     Visual C++ 48  
 resource.h 48  
 resource.h file  
     creation of 56  
     menu resource IDs 148  
 resources 55–57  
     creating 56  
     creating on Windows 56  
     for menus 147–150  
     GetXOPIndString 352  
     GetXOPNamedResource 351  
     GetXOPResource 351  
     in direct external functions 116, 118  
     in direct external operations 97  
     initialization 13  
     resource.h file 56, 148  
     STR# 1100 resource 55, 62–63  
     STR# 1101 resource 55, 220  
     STR# 1160 resource 55  
     version resources 207  
     XMI1 1100 147  
     XMI1 1100 resource 55  
     XMN1 1100 147–148  
     XMN1 1100 resource 55  
     XOP-specific 55

- XOPC 1100 resource 55, 72
- XOPF 1100 resource 55, 100
- XOPI 1100 resource 55–56
- XOPRefNum 351
- XOPSupport routines 351–352
- XOPTypes.r 56
- XPRF resources 397
- XSM1 1100 147
- XSM1 1100 resource 55
- ResourceToActualItem 151, 316
- ResourceToActualMenuID 151, 316
- result codes 61
- result types
  - in external functions 104
- results
  - external functions 103–104
  - string in external functions 108–109
- ResumeUpdate 386
- Rez
  - compiling .r files 56
- RGBColor 365
- root data folder 303
  - threads 177
- ROWS 128, 253, 281
- RS232
  - VDT2 sample XOP 12
- RunningInMainThread 180, 379
- runtime library
  - Visual C++ 50
- runtime parameter structure 79

## S

- S\_ variables
  - in external operations 94
- S\_fileName 346–347
  - SetFileLoaderOperationOutputVariables 347
- S\_path 346–347
  - SetFileLoaderOperationOutputVariables 347
- S\_waveNames 346–347
  - SetFileLoaderOperationOutputVariables 347
- sample XOPs 10–12
- SanitizeWaveName 359
- SAVE message 60
- SAVE\_TYPE\_xxx 60
- SaveIgorData 442
- SAVESETTINGS message 60, 65
- SaveXOPPrefsHandle 65, 397
- saving settings 65
- ScaleClipAndRoundData 357
- ScaleData 356
- scaling
  - waves 129
- scheme
  - Xcode 39, 430–431

- schemes
  - Xcode 39
- SDK
  - in Xcode 41
- SendWinMessageToIgor 419
- serial port
  - VDT2 sample XOP 12
- SetCurrentDataFolder 303
- SetDataFolderObject 311
- SetDataFolderProperties 301
- SetFileLoaderOperationOutputVariables 94, 347
- SetFileLoaderOutputVariables 94, 346
- SetHandleSize 199–200
- SetIgorComplexVar 296
- SetIgorFloatingVar 295
  - example 140
- SetIgorIntVar 295
- SetIgorMenuItem 153, 317
- SetIgorProcedure 371
- SetIgorStringVar 297
  - example 140
- SetNaN32 404
- SetNaN64 405
- SetNVAR 210, 298
  - example 211
- SetOperationNumVar 242
- SetOperationStrVar 243
- SetOperationStrVar2 243
- SetOperationWaveRef 87, 247
- SetPtrSize 200
- SetSVAR 210, 299
  - example 211
- SetTextWaveData 129, 288
- settings
  - GetPrefsState 398
  - GetXOPPrefsHandle 397
  - loading 65
  - messages for saving and loading settings 59–61
  - SaveXOPPrefsHandle 397
  - saving 65
  - saving and loading 65
  - structure alignment 66
  - structures 66
- SetWaveDimensionLabels 273
- SetWaveLock 258
- SetWaveNote 261
- SetWaveScaling 259
- SetWaveUnits 260
- SetXOPEntry 17, 239
  - example 53
  - XOPEntry field in IORecHandle 67
- SetXOPMessage 240
- SetXOPResult 61, 240
  - example 53
  - replace long data type 191
  - result field in IORecHandle 67

- returning custom errors 63
- SAVESETTINGS message 65
- SetXOPType 239
  - IDLE message 58
  - in external functions 116
  - XOPType field in IOREC 66
- Shift-JIS 201
- shortcuts 13
- SHOW\_MENU\_AT\_LAUNCH 148
- SHOW\_MENU\_WHEN\_ACTIVE 148
- SIGNED\_INT 354
- SimpleFit sample XOP 12, 20
- SimpleGaussFit
  - compiling 29
  - testing 30
- SimpleGaussFit sample XOP 20
- SimpleLoadWave sample XOP 12
- SInt16 data type 188
- SInt32 data type 188
- SInt64 data type 188
- SInt8 data type 188
- Slow Data Acq example 182
- solution files 48
- SpecialDirPath 333
- SpinCursor 419
- SpinProcess 214, 393
  - recursion 68
- sprintf
  - 64-bit XOPs 192
- SS\_MASK 397
- stack
  - overwriting 222
- standard names 362
- starter code 74
  - details 94
  - updating 95
- statsOp 73
- status line
  - XOPSetContextualHelpMessage 396
- StoreNumericDataUsingVarName 86, 245
- StoreNumericValue 283
- StoreNumVar 293
- StoreStringDataUsingVarName 86, 246
- StoreStrVar 294
- STR# 1100 resource 55, 62–63
- STR# 1101 resource 55, 220
  - XOP help file name 220
- STR# 1160 resource 55
- STR\_OBJECT
  - CheckName 360
  - CreateValidDataObjectName 360
- strchr2 400
- String
  - in structure parameters 209
- string expressions
  - in operations 71
- string keyword
  - Operation Handler 77
- string parameters
  - external operations 83
- string variables
  - accessing
    - XOPSupport routines 291–299
  - content stored in handles 138
  - FetchStrHandle 294
  - FetchStringDataUsingVarName 246
  - FetchStrVar 293
  - GetSVAR 298
  - SetFileLoaderOperationOutputVariables 347
  - SetIgorStringVar 297
  - SetOperationStrVar 243
  - SetOperationStrVar2 243
  - SetSVAR 299
  - StoreStringDataUsingVarName 246
  - StoreStrVar 294
  - StringList 392
  - Variable 291
  - VarNameToDataType 244
- StringList 392
- strings
  - disposing 83
  - GetCStringFromHandle 129, 138
  - in handles 233
  - in Igor 110, 138
  - in structure parameters 210
  - pitfalls 233
  - PutCStringInHandle 129, 138
  - WMGetHandleSize 233
  - WMSetHandleSize 233
- strchr2 400
- structure alignment 79, 208
  - in external functions 103
  - settings structures 66
- structure keyword
  - Operation Handler 77
- structure parameters 209
  - CheckFunctionForm 374
  - data folder reference counting 144
  - example in external function 112
  - example in external operation 90
  - extended 92
  - extended example in external operation 93
  - fields 209
  - in external functions 111
  - in external operations 89
  - NVARs 211
  - strings 210
  - SVARs 211
  - wave reference counting 133
- structures
  - expandable 65–66
  - GetNVAR 297

- GetSVAR 298
  - in XOPF resource 101–102
- SetNVAR 298
- SetSVAR 299
  - versioning 92, 114
  - wave reference counting 90, 114, 210
- submenus
  - XSM1 1100 resource 149
- SVAR 85
- SVARs
  - GetSVAR 298
  - in structure parameters 209, 211
  - SetSVAR 299
- symbolic debugging 232
- symbolic paths
  - GetFullPathFromSymbolicPathAndFilePath 345
  - GetPathInfo2 391
  - PathList 390
  - XOPFillPathMenu 325
- system text encoding 201

## T

- tableOp 73
- target extension
  - Visual C++ 49
- target machine
  - Visual C++ 51
- target name
  - Visual C++ 49
- Task Manager
  - leaks 227
- technical support 17
  - email 18
  - mailing list 18
- template 75
- templates 76
  - access waves 127
  - for operations and functions 219
  - length limit 95
- temporary storage access method 125
- text encodings 201
  - ConvertTextEncoding 402
  - in Visual C++ 204
  - in Xcode 203
  - WaveTextEncoding 261
- text utility routines 350
- text waves
  - accessing text data 128–129
  - faster access 129
  - GetTextWaveData 286
  - MDGetTextWavePointValue 284
  - MDSetTextWavePointValue 285
  - SetTextWaveData 288
  - text wave type 124
- TEXT\_WAVE\_TYPE 124, 253
  - type for waves 124
- thread groups 183–184
- thread output queue 182–183
- thread-safe
  - external functions 117
  - external operations 95
  - Igor-thread-safe 178
  - programming 175–176
- ThreadGroupGetDF 183, 380
- ThreadGroupGetDFR operation 182
- ThreadGroupPutDF 183, 380
- ThreadGroupPutDF operation 182
- threading
  - Visual C++ 49
- threads 175
  - callbacks 178
    - not allowed from private threads 179
  - CallFunction 213
  - CheckRunningInMainThread 180, 379
  - computation threads 176
  - data acquisition 180–186
  - data folders 177
  - DoXOPIdle operation 184
  - external operations 95
  - global variables 176
  - IDLE message 184
  - Igor-thread-safe 178
  - internal 176
    - callbacks not allowed 179
  - internal threads 184–185
  - main 176
  - messages from Igor 176
  - mutexes 177
    - data acquisition 184
  - parameters 177
  - private 176
    - callbacks not allowed 179
  - private threads 184–185
  - root data folder 177
  - RunningInMainThread 180, 379
  - Slow Data Acq example 182
  - thread groups 183–184
  - thread output queue 182–183
  - ThreadGroupGetDF 380
  - ThreadGroupGetDF callback 183
  - ThreadGroupGetDFR operation 182
  - ThreadGroupPutDF 380
  - ThreadGroupPutDF callback 183
  - ThreadGroupPutDF operation 182
  - ThreadStart operation 177
  - tp pointer 96, 117
  - types of 176
  - user-defined function threads 176–177
  - UserFunctionThreadInfoPtr 96, 117
  - WaveHandleModified 185



- waves 177
- XOPNotice 233
- XOPSupport routines 379–381
- threadSafeOp 73, 96
- ThreadStart operation 177
- TickCount 405
- TickCountInt data type 188
- tp pointer 96, 117
- TransformWindowCoordinates 419
- TRANSIENT bit 66
  - SetXOPType 239
- transient XOPs
  - XOPType field in IORec 66
- troubleshooting 17
- try
  - in C++ 217
- TU windows
  - in Igor Pro 7 424

## U

- UInt16 data type 188
- UInt32 data type 188
- UInt64 data type 188
- UInt8 data type 188
- UNC
  - volume names 328–329
- undo
  - menu item 317
- UnEscapeSpecialCharacters 402
- Unicode 201
  - ConvertTextEncoding 402
- uninitialized variables 222
- UniqueName 358
- UniqueName2 358
- units 259–260
  - for waves 129
  - MDGetWaveUnits 270
  - MDSetWaveUnits 271
  - waves 129
- UNSIGNED\_INT 354
- update
  - DoUpdate 386
  - XOPCommand side-effect 384
  - XOPCommand2 side-effect 384
  - XOPCommand3 side-effect 385
  - XOPSilentCommand side-effect 385
- updates for XOP Toolkit 35
- user functions
  - versus external functions 99
- user-defined function threads 176–177
- user-defined functions
  - calling from an XOP 212
- UserFunctionThreadInfoPtr 96, 117
- UTF-8 201
  - in Visual C++ 204

- in Xcode 203
- utility routines 11, 399
  - CheckRunningInMainThread 379
  - CmpStr 400
  - ConvertTextEncoding 402
  - DateToIgorDateInSeconds 405
  - EscapeSpecialCharacters 401
  - GetCStringFromHandle 399
  - IgorDateInSecondsToDate 405
  - IsINF32 404
  - IsINF64 404
  - IsNaN32 404
  - IsNaN64 404
  - MacRectToWinRect 407
  - MemClear 399
  - PutCStringInHandle 399
  - RunningInMainThread 379
  - SetNaN32 404
  - SetNaN64 405
  - strchr2 400
  - strrchr2 400
  - UnEscapeSpecialCharacters 402
  - WinInfo 396
  - WinRectToMacRect 407
  - XOPOKAlert 406
  - XOPOKCancelAlert 406
  - XOPYesNoAlert 407
  - XOPYesNoCancelAlert 407
- utilOp 73

## V

- V\_ variables
  - in external operations 94
- V\_Flag
  - SetFileLoaderOperationOutputVariables 347
- V\_flag 346–347
- VAR\_GLOBAL 291
- VAR\_OBJECT
  - CheckName 360
  - CreateValidDataObjectName 360
- Variable 291
  - in structure parameters 209
- VariableList 391
- variables
  - accessing
    - XOPSupport routines 291–299
  - accessing data in 138–140
  - commonly used XOPSupport routines 139
  - data folders 291
  - example of making 140
  - FetchNumericDataUsingVarName 244
  - FetchNumVar 292
  - FetchStrHandle 294
  - FetchStringDataUsingVarName 246
  - FetchStrVar 293

- GetNVAR 297
- GetSVAR 298
- getting in external operations 244, 246
- local versus global 291
- making global 140
- precision of 291
- SetFileLoaderOperationOutputVariables 347
- SetIgorComplexVar 296
- SetIgorFloatingVar 295
- SetIgorIntVar 295
- SetIgorStringVar 297
- SetNVAR 298
- SetOperationNumVar 242
- SetOperationStrVar 243
- SetOperationStrVar2 243
- SetSVAR 299
- setting in external operations 242–246, 347
- StoreNumericDataUsingVarName 245
- StoreNumVar 293
- StoreStringDataUsingVarName 246
- StoreStrVar 294
- StringList 392
- uninitialized 222
- VAR\_GLOBAL 291
- Variable 291
- VariableList 391
- VarNameToDataType 244
- varName keyword
  - Operation Handler 77
- VarName parameters
  - external operations 85
- VarNameToDataType 86, 244
- VDT2 sample XOP 12
- version
  - of Igor 70
- version checking 70
- version resources 207
- versions 69–70
  - checking Igor version 421
  - compatibility 4
  - of Igor 4, 72
  - of structures 92, 114
  - XOP Toolkit 7 Upgrade 415
- virtual memory
  - 64-bit XOPs 187
- Visual C++ 47–51
  - .rc files 48
  - .sln files 48
  - .vcxproj files 48
  - configurations 49
  - CR\_STR macro 421
  - creating a new project 48
  - debug configuration 49
  - debugging 51
  - first XOP support 3
  - modifying project for XOP Toolkit 7 434

- project files 48
- project settings 49
- release configuration 49
- resource files 48
- resource.h 48
- resources 56
- runtime library 50
- solution files 48
- structure alignment 208
- support for 415
- supported versions 6, 35, 47
- target extension 49
- target machine 51
- target name 49
- testing installation 8
- text encodings 204
- threading 49
- worksheet files 10
- XOPSupport project files 9
- Visual C++ 2012
  - building SimpleFit 22
- Visual C++ 9
  - building SimpleFit 22
  - testing installation 8
- Visual C++-UTF-8 204
- volume names
  - UNC 328–329
- vs keyword 76
  - optional parameters 78

## W

- warnings
  - Wno-parentheses 43
- WatchCursor 419
- wave assignment statements 103
- wave handles
  - dereferencing 230
- wave keyword
  - Operation Handler 77
- wave parameters
  - external operations 84
- wave reference counting 131
  - CLEANUP message 134
  - NEW message 134
  - structure parameters 90, 114, 133, 210
- wave reference waves 134
- WAVE references
  - data folder reference counting 133
  - DataFolderAndName parameters 87
  - destination wave example 87
  - in external operations 247–248
  - in structure parameters 209
  - SetOperationWaveRef 87
- wave references
  - automatic 87

- GetOperationDestWave 251
  - SetOperationWaveRef 247
- wave scaling 129
- wave units 129
- WAVE\_OBJECT
  - CheckName 360
  - CreateValidDataObjectName 360
- WAVE\_OBJECT object type 59
- WAVE\_TYPE 105, 124
  - curve fitting 105
  - in XOPF resource 101
  - result in external functions 106
  - wave reference parameters 105
- WaveAccess sample XOP 12, 127
- WaveData 262
  - dangling pointers 230
  - example 123
- WaveHandleModified 264
  - threads 185
- WaveList 389
- WaveLock 258
- WaveMemorySize 263
- WaveModCount 257
- WaveModDate 257
- WaveModified 264
- WaveModState 257
- WaveName 261
- WaveNote 419
- WaveNoteCopy 260
- waveOp 73
- WavePoints 256
- waveRange keyword
  - Operation Handler 77
- WaveRange parameters
  - external operations 84
- waves
  - accessing
    - XOPSupport routines 253–290
  - accessing data in 121–137
  - accessing numeric data 125–128
  - accessing text data 128–129
  - accessing with C++ templates 127
  - CalcWaveRange 264
  - ChangeWave 254
  - commonly used XOPSupport routines 121
  - containing data folder references 135
  - containing wave references 134
  - data folders 306
  - data scaling 129
  - destination wave
    - SetOperationWaveRefs 247
  - destination waves 87
  - dimension scaling 129
  - direct access method 126
  - example making 123
  - FetchNumericValue 283
  - FetchWave 255
  - FetchWaveFromDataFolder 255
  - FileLoaderMakeWave 346
  - free 135
    - GetOperationDestWave 250
    - MDMakeWave 266
  - GetOperationWaveRef 248
  - GetTextWaveData 286
  - GetWaveDimensionLabels 272
  - HoldWave 289
  - killing 130
  - killing versus deleting 131
  - KillWave 255
  - lock state 258
  - MakeWave 253
  - MDAccessNumericWaveData 273
  - MDChangeWave 267
  - MDChangeWave2 268
  - MDGetDimensionLabel 271
  - MDGetDPDataFromNumericWave 280
  - MDGetNumericWavePointValue 276
  - MDGetNumericWavePointValueSInt64 278
  - MDGetNumericWavePointValueUInt64 279
  - MDGetTextWavePointValue 284
  - MDGetWaveDimensions 266
  - MDGetWaveScaling 269
  - MDGetWaveUnits 270
  - MDMakeWave 265
  - MDSetDimensionLabel 272
  - MDSetNumericWavePointValue 276
  - MDSetNumericWavePointValueSInt64 278
  - MDSetNumericWavePointValueUInt64 280
  - MDSetTextWavePointValue 285
  - MDSetWaveScaling 269
  - MDSetWaveUnits 271
  - MDStoreDPDataInNumericWave 281
  - name length limits 206
  - number types 124
  - OBJINUSE message 59
  - organization of numeric data 127
  - parameters in external functions 105
  - point access method 125
  - preventing killing 130
  - ranges of 84
  - reference counting 131
  - ReleaseWave 289
  - results in external functions 106
  - SanitizeWaveName 359
  - scaling 129
  - SetFileLoaderOutputVariables 346
  - SetOperationWaveRef 247
  - SetTextWaveData 288
  - SetWaveDimensionLabels 273
  - SetWaveLock 258
  - SetWaveNote 261
  - SetWaveScaling 259

- SetWaveUnits 260
- speed of accessing numeric data 127
- StoreNumericValue 283
- temporary storage access method 125
- text wave type 124
- threads 177
- units 129
- WaveAccess example 12
- WaveData 262
- WaveHandleModified 264
- WaveList 389
- WaveLock 258
- WaveMemorySize 263
- WaveModCount 257
- WaveModDate 257
- WaveModified 264
- WaveModState 257
- WaveName 261
- WaveNoteCopy 260
- WavePoints 256
- WaveScaling 258
- WaveTextEncoding 261
- WaveType 256
- WaveUnits 259
- XOPFillWaveMenu 325
- XOPWaveAccess.c 9
- WaveScaling 258
- WaveTextEncoding 261
- WaveType 256
- WaveUnits 259
- Windows
  - DLLs 217
- windows
  - getting Igor client HWND 382
  - text
    - XOPSupport routines 350
  - WinList 390
  - XOPFillWinMenu 326
  - XOPSetContextualHelpMessage 396
  - XOPSupport routines 349
- Windows OS
  - IgorClientHWND 382
  - IgorModule 382
  - WindowsErrorToIgorError 383
  - WMGetLastError 382
  - XOPModule 382
- Windows OS error codes 62
- Windows platform
  - compatibility with Igor Pro 5 3
  - first Igor release 3
  - supported OS versions 6
  - XOPSupport routines 382–383
- Windows-1252 201
- WindowsErrorToIgorError 383
- WINIGOR #define 418
- WinInfo 396
- WinList 390
- WinRectToMacRect 407
- WinToMacPath 206, 329
- WM memory routines 410–413
  - WMDisposeHandle 412
  - WMDisposePtr 413
  - WMGetHandleSize 410
  - WMGetPtrSize 412
  - WMHandAndHand 411
  - WMHandToHand 411
  - WMNewHandle 410
  - WMNewPtr 412
  - WMPtrAndHand 413
  - WMPtrToHand 413
  - WMSetHandleSize 410
  - WMSetPtrSize 412
- WM memory XOPSupport Routines
  - updating old code 199
- WM memory XOPSupport routines 195–201
  - WMDisposeHandle 200
  - WMDisposePtr 200
  - WMGetHandleSize 200
  - WMGetPtrSize 200
  - WMHandAndHand 200
  - WMHandToHand 200
  - WMNewHandle 200
  - WMNewPtr 200
  - WMPtrAndHand 200
  - WMPtrToHand 200
  - WMSetHandleSize 200
  - WMSetPtrSize 200
- WM\_COMMAND message 150
- WM\_INITMENU message 152
- WMDisposeHandle 200, 412
- WMDisposePtr 200, 413
- WMGetHandleSize 110, 138, 200, 410
  - don't use on wave handles 263
  - use with strings 233
- WMGetLastError 382
- WMGetPtrSize 200, 412
- WMHandAndHand 200, 411
- WMHandToHand 200, 411
- WMNewHandle 197, 200, 410
  - example 197
- WMNewPtr 200, 412
- WMPtrAndHand 200, 413
- WMPtrToHand 200, 413
- WMSetHandleSize 198, 200, 410
  - dangling pointers 230
  - use with strings 233
- WMSetPtrSize 200, 412
- WMTypes.h 190
- WMTypes.h file 188
- worksheet files 10
- World-Wide Web 18

# X

X scaling 258–259

MDGetWaveScaling 269

MDSetWaveScaling 269

x86 3

Xcode 35

.exp files 36, 42

.Info.plist 42

architectures 42

base SDK 41

build location 39

bundles 37

compiler version 41

compiling for Intel 25

creating a new project 36

creating project in guided tour 25

debugging 46

deployment target 41

executable name 43

frameworks 37

IGR0 XOP file creator 43

Info.plist 37

Info.plist file 43

InfoPlist.strings file 43

IXOP XOP file type 43

LLVM compiler 4

Mach-O 36

packages 36

prefix files 37, 42

project settings 38, 40

Project Settings dialog 38

scheme 39, 430–431

schemes 39

SDK 41

support for 415

supported operating systems 6

supported versions 6, 35

testing installation 7

text encodings 203

updating projects for XOP Toolkit 7 427

UTF-8 203

XOPSupport project files 9

Xcode 10 4

updating projects for 44, 431

Xcode 4

building SimpleFit 20

compiler version 4

LLVM 4

Xcode 9

updating projects for 427

XFUNC1 sample XOP 11

XFUNC1 XOP

description 99

direct functions 116

XOPF 1100 resource example 100

XFUNC1Add

example external function 99

XFUNC1ComplexConjugate

example external function 99, 108

XFUNC1Div

example external function 99

invoking 103

parameter types 104

parameters and result 103

XFUNC2 sample XOP 11

XFUNC2 XOP

description 99

direct functions 116

XOPF 1100 resource 104

XFUNC3 sample XOP 11

XFUNC3 XOP

description 100

XOPF 1100 resource 108

XMI1 1100 resource 55, 147–148

itemFlags field 152

XMN1 1100 resource 55, 147–148

XOP errors 62

XOP files 13

XOP protocol 13–14

compatibility 69

GetXOPItem 241

GetXOPMessage 240

GetXOPResult 240

GetXOPStatus 241

SetXOPEntry 239

SetXOPMessage 240

SetXOPResult 240

SetXOPType 239

version 57

XOPInit 239

XOP protocol version 70

XOP resources 55–57

XOP Toolkit

installing 7

overview 8–10

sample XOPs 10

updates 35

version 57

version 6 4

version 7 upgrade 415

XOP Toolkit 6 3

64-bit XOPs 187

changes to main function 421

programming for 64-bit XOPs 189

supported Igor versions 421

XOP Toolkit 6.3 4

XOP Toolkit 7

changed XOPSupport routines 419

header file changes 418

miscellaneous changes 421

new XOPSupport routines 420

- porting existing XOPs to 416
- removed XOPSupport routines 419
- text encoding 418
- updating Visual C++ project 434
- updating Xcode project 427
- XOP Toolkit number type codes 354, 356
- XOP windows
  - Cocoa 155, 165
  - Objective-C++ 156, 165
- XOP\_FEATURE\_FLAGS 56, 442
- XOP\_TOOLKIT\_VERSION 56–57
- XOP\_VERSION 56–57, 70
- XOP.h 9
- XOP1 sample XOP 11
- XOPActualMenuIDToMenuRef 318
- XOPAppendMenuItem 321
- XOPAtEndOfFile 338
- XOPC 1100 resource 53, 55, 72
- XOPCheckMenuItem 324
- XOPCloseFile 335
- XOPCommand 384
  - recursion 68
- XOPCommand2 384
- XOPCommand3 385
- XOPCountMenuItems 319
- XOPCreateFile 334
- XOPDeleteFile 334
- XOPDeleteMenuItem 322
- XOPDeleteMenuItemRange 322
- XOPDisableMenuItem 323
- XOPDisplayHelpTopic 220, 395
- XOPEnableMenuItem 323
- XOPEntry 13, 17, 55, 57, 67
  - extern "C" declaration 55
  - FUNCTION message 116
  - messages 13
  - replace long data type 191
- XOPF 1100 resource 55, 116
  - complex parameters 108
  - data folder reference parameters 106
  - defined 100
  - inspected by Igor Pro 100
  - string parameters 108
  - wave parameters 105
  - XFUNC2 XOP example 104
- XOPFillMenu 317
- XOPFillMenuNoMeta 324
- XOPFillPathMenu 325
- XOPFillWaveMenu 325
- XOPFillWinMenu 326
- XOPGetClipboardData 408
- XOPGetFilePosition 337
- XOPGetFilePosition2 338
- XOPGetMenuInfo 319
- XOPGetMenuItemInfo 320
- XOPGetMenuItemText 321
- XOPHideMainMenu 320
- XOPI 1100 resource 53, 55–56
  - XOP protocol version 70
- XOPI\_RESERVED 56, 442
- XOPInit 17, 239
  - example 53
  - XOPRecHandle global 66
- XOPInsertMenuItem 322
- XOPIORecParam data type 192
- XOPIORecResult data type 191
- XOPMain 195
  - receives IORecHandle 17
- XOPMenuRef 147
- XOPModule 382
- XOPNotice 386
  - for debugging 232
  - from a thread 233
  - pitfalls 233
- XOPNotice2 387
- XOPNotice3 388
- XOPNumberOfBytesInFile 339
- XOPNumberOfBytesInFile2 339
- XOPOKAlert 64, 406
- XOPOKCancelAlert 64, 406
- XOPOp
  - operation categories
  - XOPOp 73
- XOPOp operation category 73
- XOPOpenFile 334
- XOPOpenFileDialog 207, 339
  - using XOPOpenFileDialog2 instead 420
- XOPOpenFileDialog2 341
  - using instead of XOPOpenFileDialog 420
- XOPReadFile 335
- XOPReadFile2 335
- XOPReadFile64 336
- XOPReadLine 336
- XOPRecHandle global 66
- XOPRefNum 351
- XOPResNotice 389
- XOPResourceMenuIDToMenuRef 318
- XOPs
  - activating 6
- XOPS 1100 resource 65
- XOPSaveFileDialog 207, 342
  - changes in XOP Toolkit 7 420
- XOPSaveFileDialog2 344
- XOPSetContextualHelpMessage 396
- XOPSetFilePosition 338
- XOPSetFilePosition2 338
- XOPSetMenuItemText 321
- XOPShowMainMenu 319
- XOPSilentCommand 385
  - recursion 68
- XOPStandardHeaders.h 9
  - changes in XOP Toolkit 7 418

- XOPStructureAlignmentTwoByte.h 79
- XOPSupport
  - callbacks 11, 13
  - header file changes 418
  - new routines 420
  - removed routines 419
  - supported Igor versions 421
  - utility routines 11
  - XOPSupport folder 11
- XOPSupport 7
  - changed routines 419
  - text encoding 418
- XOPSupport folder 9, 11
- XOPSupport project files 9
- XOPSupport routines 238–413
  - color table routines 364–365
  - command window 366–369
  - for accessing data folders 300–315
  - for accessing files 328–344
  - for accessing variables 291–299
  - for accessing waves 253–290
  - for communicating with Igor 239–241
  - for dialogs 327
  - for FIFOs 353
  - for file loaders 345–348
  - for menus 316–326
  - for text windows 350
  - for windows 349
  - miscellaneous 384–398
  - multithreading support routines 379–381
  - numeric conversion 354–357
  - object names 358–363
  - Operation Handler 242
  - procedures 370–378
  - programming utilities 399–409
  - resources 351–352
  - Windows-specific routines 382–383
- XOPSupport.c 9
- XOPSupport.h 9
- XOPTypes.r 56
- XOPWaveAccess.c 9
- XOPWindows.h 418
- XOPWriteFile 337
- XOPWriteFile64 337
- XOPYesNoAlert 64, 407
- XOPYesNoCancelAlert 64, 407
- XPRF resources 397
- XSM1 1100 resource 55, 147
- xstrcat
  - example external function 100, 109
  - XOPF 1100 resource 108
- xstrcat0 external function 11
- xstrcat1 external function 11

