



# iTool Developer's Guide

**RSI**  
Research Systems Inc.

IDL Version 6.0  
July, 2003 Edition  
Copyright © Research Systems, Inc.  
All Rights Reserved

## Restricted Rights Notice

The IDL<sup>®</sup>, ION Script<sup>™</sup>, and ION Java<sup>™</sup> software programs and the accompanying procedures, functions, and documentation described herein are sold under license agreement. Their use, duplication, and disclosure are subject to the restrictions stated in the license agreement. Research Systems, Inc., reserves the right to make changes to this document at any time and without notice.

## Limitation of Warranty

Research Systems, Inc. makes no warranties, either express or implied, as to any matter not expressly set forth in the license agreement, including without limitation the condition of the software, merchantability, or fitness for any particular purpose.

Research Systems, Inc. shall not be liable for any direct, consequential, or other damages suffered by the Licensee or any others resulting from use of the IDL or ION software packages or their documentation.

## Permission to Reproduce this Manual

If you are a licensed user of this product, Research Systems, Inc. grants you a limited, nontransferable license to reproduce this particular document provided such copies are for your use only and are not sold or distributed to third parties. All such copies must contain the title page and this notice page in their entirety.

## Acknowledgments

IDL<sup>®</sup> is a registered trademark and ION<sup>™</sup>, ION Script<sup>™</sup>, ION Java<sup>™</sup>, are trademarks of Research Systems Inc., registered in the United States Patent and Trademark Office, for the computer program described herein.

Numerical Recipes<sup>™</sup> is a trademark of Numerical Recipes Software. Numerical Recipes routines are used by permission.

GRG2<sup>™</sup> is a trademark of Windward Technologies, Inc. The GRG2 software for nonlinear optimization is used by permission.

NCSA Hierarchical Data Format (HDF) Software Library and Utilities  
Copyright 1988-2001 The Board of Trustees of the University of Illinois  
All rights reserved.

NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities  
Copyright 1998, 1999, 2000, 2001, 2002 by the Board of Trustees of the University of Illinois. All rights reserved.

CDF Library  
Copyright © 1999  
National Space Science Data Center  
NASA/Goddard Space Flight Center

NetCDF Library  
Copyright © 1993-1996 University Corporation for Atmospheric Research/Unidata

HDF EOS Library  
Copyright © 1996 Hughes and Applied Research Corporation

This software is based in part on the work of the Independent JPEG Group.

Portions of this software are copyrighted by INTERSOLV, Inc., 1991-1998.

Use of this software for providing LZW capability for any purpose is not authorized unless user first enters into a license agreement with Unisys under U.S. Patent No. 4,558,302 and foreign counterparts. For information concerning licensing, please contact: Unisys Corporation, Welch Licensing Department - C1SW19, Township Line & Union Meeting Roads, P.O. Box 500, Blue Bell, PA 19424.

Portions of this computer program are copyright © 1995-1999 LizardTech, Inc. All rights reserved. MrSID is protected by U.S. Patent No. 5,710,835. Foreign Patents Pending.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>)

IDL Wavelet Toolkit Copyright © 2002 Christopher Torrence.

Other trademarks and registered trademarks are the property of the respective trademark holders.



# Contents

<b>Chapter 1:</b>	
<b>Overview</b> .....	<b>7</b>
What are iTools? .....	8
What is the iTools Component Framework? .....	9
About this Manual .....	10
About the iTools Code Base .....	11
Skills Required to Use the iTools Component Framework .....	13

## **Part I: Understanding the iTools Component Framework**

<b>Chapter 2:</b>	
<b>iTool System Architecture</b> .....	<b>17</b>
Overview .....	18
iTool Object Identifiers .....	19
iTool Object Hierarchy .....	21
Registering Components .....	22

iTool Messaging System .....	25
System Resources .....	28

### **Chapter 3: Data Management ..... 31**

Overview .....	32
iTool Data Manager .....	33
iTool Data Types .....	34
iTool Data Objects .....	36
Predefined iTool Data Classes .....	38
Parameters .....	41
Data Type Matching .....	43
Data Update Mechanism .....	45

### **Chapter 4: Property Management ..... 47**

About the Properties Interface .....	48
Property Data Types .....	51
Registering Properties .....	54
Property Identifiers .....	57
Property Attributes .....	58
Property Aggregation .....	61
Property Update Mechanism .....	63
Properties of the iTools System .....	64

## **Part II: Using the iTools Component Framework**

### **Chapter 5: Creating an iTool ..... 67**

Overview .....	68
Creating a New iTool Class .....	69
Registering a New Tool Class .....	78
Creating an iTool Launch Routine .....	80
Example: Simple iTool .....	85

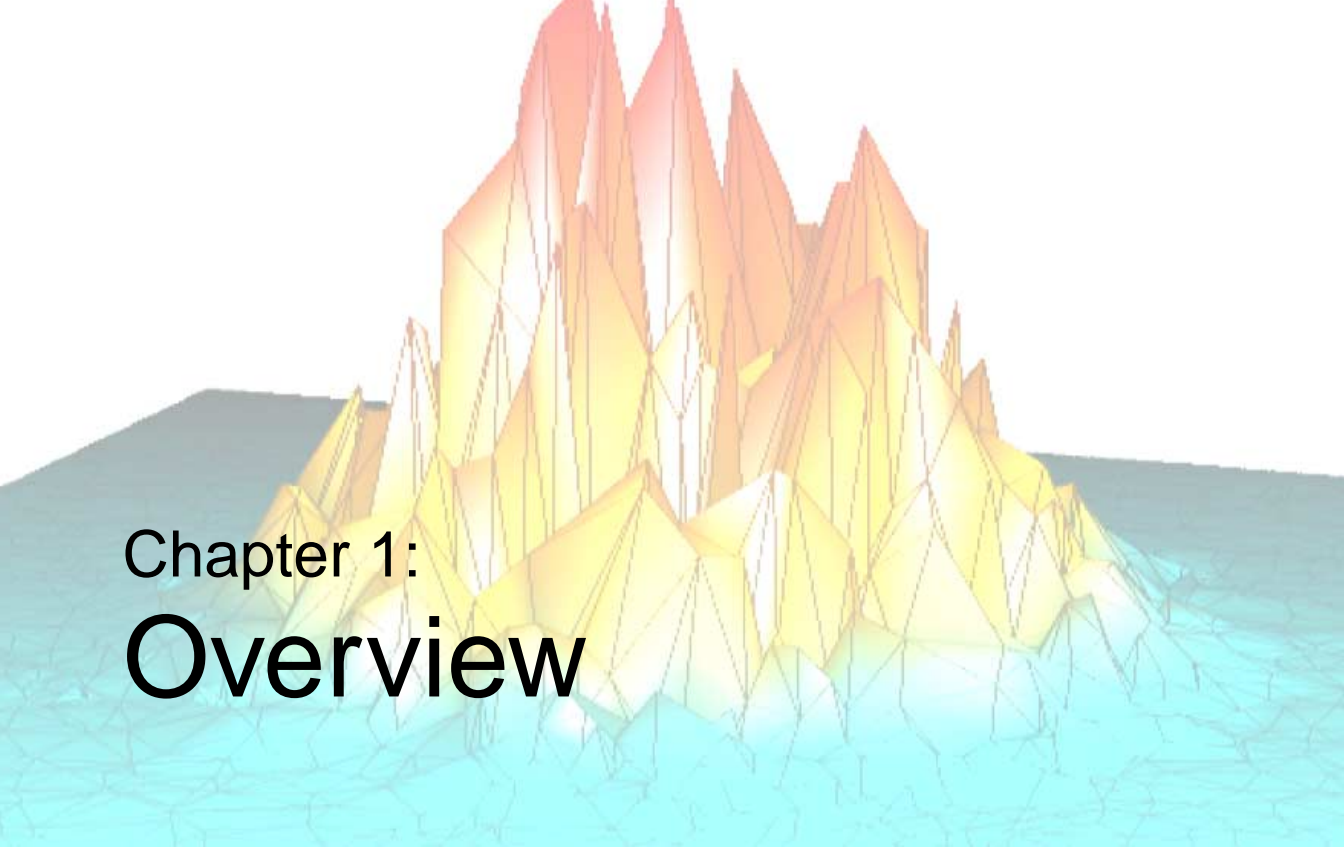
### **Chapter 6: Creating a Visualization ..... 89**

Overview .....	90
----------------	----

Predefined iTool Visualization Classes .....	91
Creating a New Visualization Type .....	95
Registering a Visualization Type .....	110
Unregistering a Visualization Type .....	112
Example: Image-Contour Visualization .....	113
<b>Chapter 7:</b>	
<b>Creating an Operation .....</b>	<b>119</b>
Overview .....	120
Predefined iTool Operations .....	122
Operations and the Undo/Redo System .....	123
Creating a New Data-Centric Operation .....	125
Creating a New Generalized Operation .....	138
Registering an Operation .....	153
Unregistering an Operation .....	155
Example: Data Resample Operation .....	156
<b>Chapter 8:</b>	
<b>Creating a File Reader .....</b>	<b>161</b>
Overview .....	162
Predefined iTool File Readers .....	163
Creating a New File Reader .....	166
Registering a File Reader .....	177
Unregistering a File Reader .....	178
Example: TIFF File Reader .....	179
<b>Chapter 9:</b>	
<b>Creating a File Writer .....</b>	<b>185</b>
Overview .....	186
Predefined iTool File Writers .....	187
Creating a New File Writer .....	190
Registering a File Writer .....	201
Unregistering a File Writer .....	202
Example: TIFF File Writer .....	203

## Part III: Modifying the iTool User Interface

<b>Chapter 10:</b>	
<b>iTool User Interface Architecture .....</b>	<b>209</b>
Overview .....	210
User Interface Objects .....	212
<b>Chapter 11:</b>	
<b>Using iTool User Interface Elements .....</b>	<b>215</b>
Overview .....	216
Status Messages .....	217
Prompts .....	219
Informational Messages .....	221
<b>Chapter 12:</b>	
<b>Creating a User Interface Service .....</b>	<b>223</b>
Overview .....	224
Predefined iTool UI Services .....	225
Creating a New UI Service .....	226
Registering a UI Service .....	231
Executing a User Interface Service .....	233
Example: Changing a Property Value .....	234
<b>Chapter 13:</b>	
<b>Creating a User Interface Panel .....</b>	<b>241</b>
Overview .....	242
Creating a UI Panel Interface .....	243
Creating Callback Routines .....	248
Registering a UI Panel .....	250
Example: A Simple UI Panel .....	252
<b>Index .....</b>	<b>261</b>



# Chapter 1: Overview

This chapter provides an overview of the IDL iTool Component Framework.

---

<a href="#">What are iTools? . . . . .</a>	<a href="#">8</a>	<a href="#">About the iTools Code Base . . . . .</a>	<a href="#">11</a>
<a href="#">What is the iTools Component Framework? .</a>	<a href="#">9</a>	<a href="#">Skills Required to Use the iTools Component Framework . . . . .</a>	<a href="#">13</a>
<a href="#">About this Manual . . . . .</a>	<a href="#">10</a>		

# What are iTools?

IDL *Intelligent Tools*, or *iTools*, are applications written in IDL to perform a variety of data analysis and visualization tasks. iTools share a common underlying application framework, presenting a full-featured, customizable, application-like user interface with menus, toolbars, and other graphical features. Several pre-defined iTools are provided along with IDL; you can use these tools to explore and visualize your data without writing any new code yourself. For information on using the standard iTools provided with IDL, see the *iTool User's Guide*.

But iTools are more than just a set of pre-written IDL programs. Behind the iTool system lies the IDL Intelligent Tools Component Framework — a set of object class files and associated utilities designed to allow you to easily extend the supplied toolset or create entirely new tools of your own. This manual will help you understand the iTools Component Framework so that you can customize existing iTools or create entirely new ones.



# What is the iTools Component Framework?

The iTools component framework is a set of object class definitions written in the IDL language. It is designed to facilitate the development of sophisticated visualization tools by providing a set of pre-built components that provide standard features including:

- creation of visualization graphics
- mouse manipulations of visualization graphics
- annotations
- management of visualization and application properties
- undo/redo capabilities
- data import and export
- printing
- data filtering and manipulation
- interface element event handling

In addition, the iTools component framework makes it easy to extend the system with components of your own creation, allowing you to design a tool to manipulate and display your data in any way you choose.

## Advantages of Using the Framework

If you are accustomed to creating user interfaces for your IDL applications using IDL widgets, using the iTools component framework will shorten your development time by providing much of the application interface via the standard component building blocks. In many cases, you are freed entirely from the need to create your own interface elements, handle widget events, and manage the display of data. Even when your application calls for additional user interface elements, the framework eliminates the need for you to manually create those elements that your application has in common with the standard iTool interface.

If you are accustomed to using IDL object graphics in your applications, the iTools component framework provides a streamlined way of working with the object graphics hierarchy. Many tasks, such as management of object properties and manipulation of the object model, are handled automatically.

# About this Manual

The *iTool Developer's Guide* describes the IDL iTools component framework and provides examples of its use. After reading this manual, you will understand how to use the component framework to create your own intelligent tools.

This manual is divided into three parts:

## Part I: Understanding the iTools Component Framework

This section describes the iTools component framework in conceptual terms, and outlines some of the processes you will use in creating new tools using the framework. While an understanding of the topics in this section may be beneficial as you develop your own applications, a complete understanding of the way the framework operates is not required to begin building your own tools.

## Part II: Using the iTools Component Framework

This section walks you through the process of creating a new iTool application, either by extending an existing iTool or by building a new tool from scratch.

## Part III: Modifying the iTool User Interface

This section discusses the process of adding your own interface elements to an iTool application.

## What this Manual is Not

This manual is not an API reference for the iTools object classes. Reference documentation for the iTool classes, methods, and properties is located in the *IDL Reference Guide*.

This manual is *not* a complete description of the object classes that constitute the iTools component framework. We describe the object classes you will use to create new iTools, but not necessarily the building blocks from which those classes are constructed. If you desire a deeper understanding of how the component framework functions than this manual provides, you can inspect the object class definition files, which are provided in IDL `.pro` source code format in the `itools/framework` subdirectory of your IDL `lib` directory.

See “[Documented vs. Undocumented Classes](#)” on page 11 for a complete explanation of our approach to documenting the iTool component framework.

# About the iTools Code Base

The iTools component framework is written almost entirely in the IDL language. The IDL code that implements both the component framework and all of the standard iTools included with IDL is available for you to inspect, copy, and learn from.

To inspect the iTools code, look in the `lib/itools` subdirectory of your IDL installation directory. The iTools code base is organized as follows:

- In the `lib/itools` directory you will find code that implements the iTool launch routines. These routines can be called directly at the IDL command line to launch a specific iTool.
- In the `lib/itools/framework` directory you will find the core iTool object class definitions and utility routines. The classes in this directory define how the iTools operate; they are made available for your inspection, but they should not be altered.
- In the `lib/itools/components` directory you will find derived iTool object classes. The classes in this directory implement the non-core features of the iTool toolset as included with IDL. You are encouraged to use these classes to implement your own iTool functionality, either by subclassing from a derived iTool object class or by modifying a copy of the class definition for a derived class.
- In the `lib/itools/ui_widgets` directory you will find the IDL code that creates an iTool user interface using IDL widgets. You may find it useful to inspect some of these routines if you are creating a side panel or a dialog used to collect parameter settings for an operation. See [Chapter 10, “iTool User Interface Architecture”](#) for additional information on creating additional user interfaces for an iTool.

## Documented vs. Undocumented Classes

If you inspect the `lib/itools` directory and its subdirectories, you will notice that there are many more classes included in the iTools component framework than are documented in the *IDL Reference Guide* and in this manual. Our approach to documenting the iTools code that is included with IDL is as follows:

- iTool launch routines for iTools included in the IDL distribution are documented in the *IDL Reference Guide*. Use of the launch routines for the pre-built iTools is discussed in the *iTool User’s Guide*.

- The core iTool component framework classes used to build individual iTools, visualization types, operations, *etc.* are formally documented in the *IDL Reference Guide* and discussed in detail in this manual. If an object class, method, or property is necessary for the construction of a new iTool or component of an iTool, it is formally documented in the *IDL Reference Guide* or in this manual. Core iTool framework classes are located in the `lib/itools/framework` subdirectory of the IDL installation directory.
- Supporting iTool component framework classes — those used to implement the documented component framework classes — are not formally documented. As noted previously, the code for these classes is available for inspection. Supporting iTool framework classes are located in the `lib/itools/framework` subdirectory of the IDL installation directory.
- Derived iTool classes — those used to implement individual iTools and their features — are not formally documented. These classes are derived from the formally documented classes, and as such can be understood by referring to the formal documentation. Derived iTool framework classes are located in the `lib/itools/components` subdirectory of the IDL installation directory.
- iTool user interface routines are not formally documented. These routines use standard IDL widget programming techniques, and as such can be understood by referring to the IDL widget documentation. User interface routines are located in the `lib/itools/ui_widgets` subdirectory of the IDL installation directory.

## Warning on Using Undocumented Features

While you are encouraged to inspect the iTools code, and to copy or subclass from derived classes and user interface routines, be aware that classes and routines that are not formally documented are not guaranteed to remain the same from one release of IDL to the next. Keep the following points in mind when implementing your own iTools:

- RSI will change undocumented supporting classes as necessary to improve the iTools system.
- RSI may also change undocumented derived classes to fix problems or add functionality; in these cases, we will make every effort to preserve backwards compatibility, but this is not guaranteed.

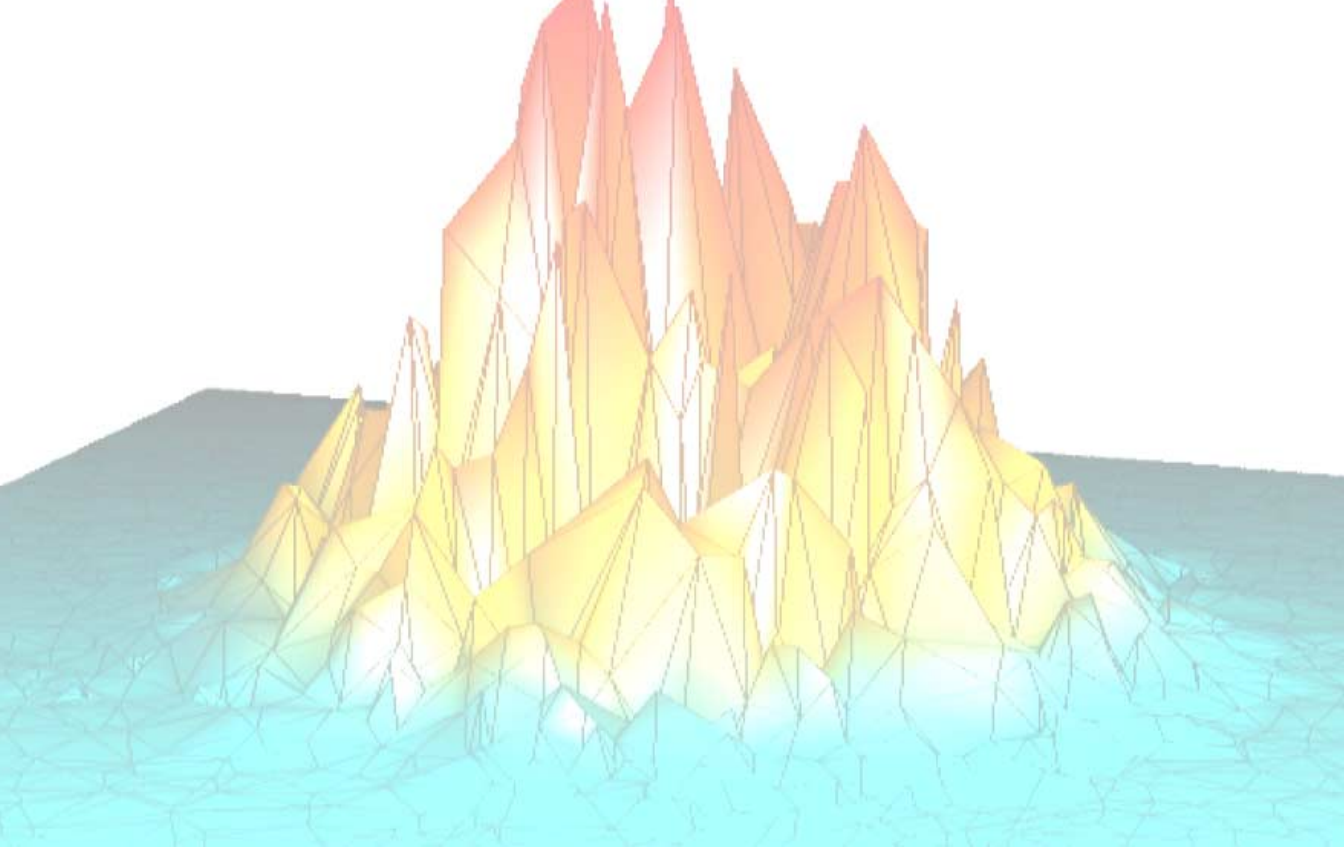
If you create new iTool classes based only on the formally documented iTool interfaces, your tools should operate properly with future releases of IDL. If you base your tools on undocumented derived classes, minor modifications *may* be necessary to ensure future compatibility.

# Skills Required to Use the iTools Component Framework

The iTools component framework consists of a set of IDL object classes, supplemented by utility routines. If you are already familiar with the concepts of object-oriented programming, or have written programs that use IDL object graphics, you will find the iTools framework easy to understand and use. The framework approach means that most of the details of creating a full-featured and usable application are already taken care of, leaving you free to concentrate on how best to manipulate and visualize your data.

If you are familiar with procedural programming in IDL but new to object-oriented programming, you will find developing iTools to be a gentle introduction to the topic. The iTools framework has been designed to allow IDL users with little or no experience writing object-oriented programs to easily customize and extend the basic iTool applications. While some familiarity with the concepts of object-oriented programming is necessary to successfully develop iTools, you should be able to create simple modifications of existing tools almost immediately, and more complex customizations soon thereafter.

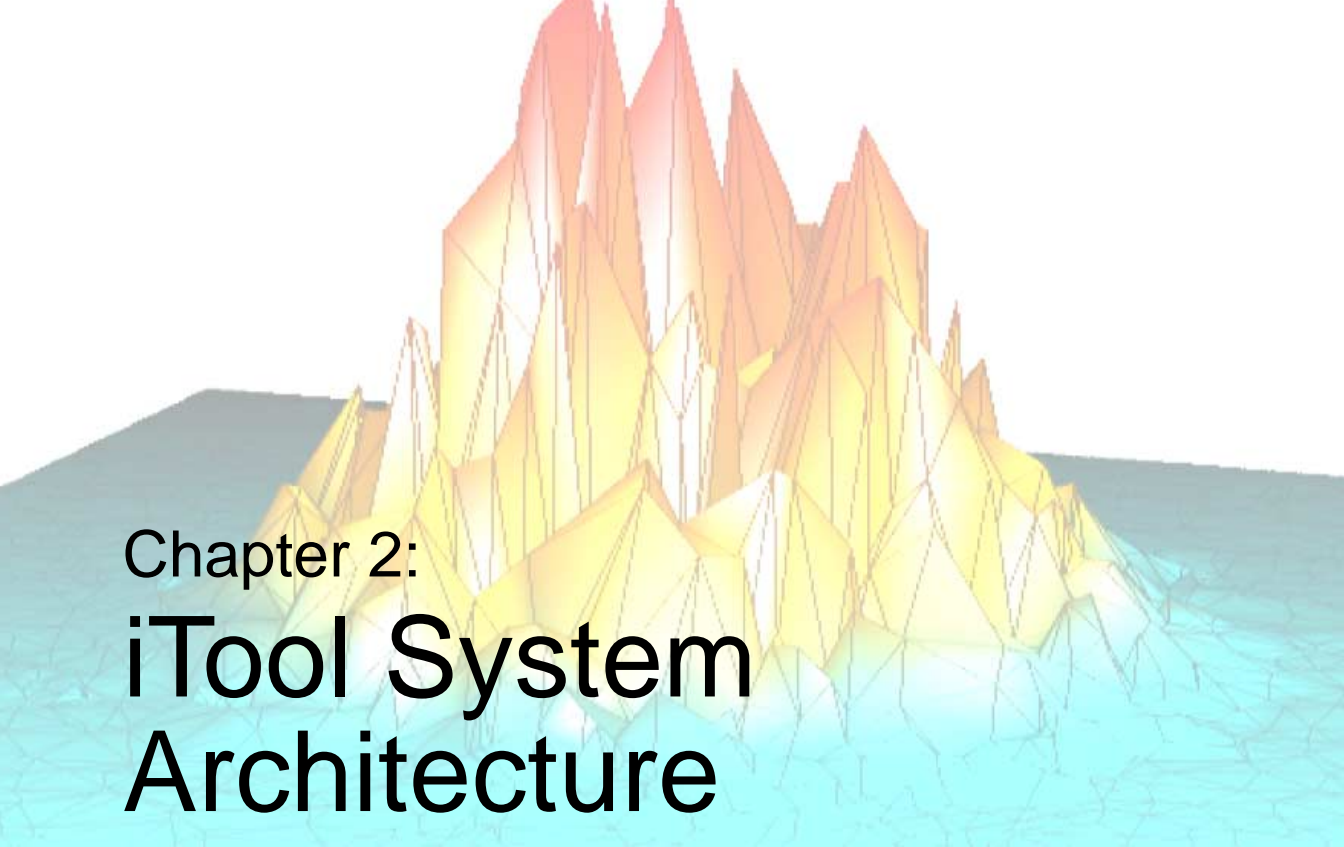




# ***Part I: Understanding the iTools Component Framework***







# Chapter 2: iTool System Architecture

This chapter describes the iTool component framework architecture.

---

<a href="#">Overview</a> .....	18	<a href="#">Registering Components</a> .....	22
<a href="#">iTool Object Identifiers</a> .....	19	<a href="#">iTool Messaging System</a> .....	25
<a href="#">iTool Object Hierarchy</a> .....	21	<a href="#">System Resources</a> .....	28

# Overview

The iTool system architecture is designed to maintain a separation between the *functionality* provided by an iTool and the graphical *presentation layer* that reveals that functionality to an iTool user (the iTool user interface). Such a separation allows for the creation of different user interfaces for the same underlying functionality; while the initial iTool user interface has been created using IDL widgets, it is easy to imagine using other technologies to create an interface to the underlying iTool functionality.

To support the goal of enabling different user interfaces for a given set of iTool functionality, the iTool architecture includes the following features:

- A design in which a single iTool object (based on the IDLitTool class) contains all non-interactive tool functionality. Similarly, a single iTool object (based on the IDLitUI class) contains all user interface functionality.
- An *object identifier* system that provides a platform-neutral way to identify objects across process and machine boundaries. Additionally, the object identifier system is designed to work with existing component technologies such as COM and Java.
- A minimal connection between the non-interactive tool functionality and the presentation layer. The tool architecture provides a small set of highly abstract methods that the tool and presentation layer use to communicate with each other. This minimal connection means that the presentation layer needs only a single object reference to the iTool object itself.
- A *messaging* system that allows one component to observe another, receiving *notification messages* when the observed component changes in some way.

This chapter describes some of the core ideas of the iTool system: object identifiers, the iTool system object and the object hierarchy it contains, the concept of registration, and how information is passed between iTool components.

# iTool Object Identifiers

iTool *object identifiers* are simple strings that uniquely identify individual objects within the hierarchy of iTool objects in much the same way that a computer file system identifies files within a hierarchy of files. The object hierarchy (and, by extension, the object identifiers) also describe where information about objects is made visible in the iTool user interface; see [“iTool Object Hierarchy”](#) on page 21 for additional discussion of the iTool hierarchy and the iTool system object.

Besides providing a familiar, user-readable way to identify objects in the iTool system, object identifiers also allow iTool developers to refer to an object without having to maintain an actual object reference to that object. This ability to use a lightweight string object to refer to a potentially “heavy” object in the iTool system makes it possible to maintain a very loose coupling between the objects that implement an iTool’s functionality and those that implement its user interface. While the iTools included in the initial release of the iTool system do not rely on this loose coupling between functionality and user interface, it allows for object access that can cross process and machine boundaries, paving the way for the use of the iTool system in more distributed environments.

---

**Note**

Object identifiers are not to be confused with *object descriptors*. See [“Object Descriptors”](#) on page 20 for details.

---

Object identifier strings are assigned when an object class is registered with either an individual iTool or with the iTool system object. See [“Registering Components”](#) on page 22 for a discussion of the registration process.

Identifiers can either be *fully qualified*, meaning that they depict the entire path from the root iTool system object to the object being identified, or *relative*, meaning they depict the path from the root of the current iTool. Fully qualified identifiers begin with the “/” character, and refer to objects that are accessible to all iTools that become active during the lifetime of the iTool system object. Relative identifiers do not begin with a “/” and refer to objects that are accessible only to the current iTool.

For example, the identifier string

```
/DATA MANAGER/MY DATA
```

refers to an object named `MY DATA`, located in the system-level `DATA MANAGER` container. Because the identifier is fully qualified, the `MY DATA` object is visible to any iTool that is active during the iTool session.

Similarly, the identifier string

```
OPERATIONS/FILTERS/MY_FILTER
```

refers to an object named `MY_FILTER`, located in a sub-container of the iTool-level `OPERATIONS` container named `FILTERS`. Because the identifier is relative, the `MY_FILTER` object is visible only to the current iTool.

---

**Note**

Object identifiers are stored as upper-case strings. Spaces are allowed.

---

## Proxy Identifiers

In some cases, you want the same object to be located in multiple places in the iTool object hierarchy. For example, the Undo operation appears in two places in the standard iTool user interface: under the **Edit** menu and on the toolbar. Rather than duplicating the Undo operation object in each of those places in the iTool object hierarchy, we can use a *proxy* mechanism to register the same object instance with multiple object identifiers. In the case of the Undo operation, the operation itself is located in the `EDIT` subcontainer of the iTool's `OPERATIONS` container, which implies that the operation appears under the iTool's **Edit** menu. A proxy (or alias) to this object is created in the `EDIT` subcontainer of the iTool's `TOOLBAR` container, which places the operation on the toolbar as well. Only one instance of the Undo object is created, but its action can be invoked from both the menu and the toolbar.

Proxy identifiers are assigned by the `Register` method for the object being proxied. See [“Registering Components”](#) on page 22 for additional details.

## Object Descriptors

*Object descriptors* are iTool objects that contain enough information about a given object class to create an object of that class when necessary. In many cases, object descriptors, rather than instances of the objects they create, are stored in the iTool hierarchy; this approach allows object instances to be created only when needed. Object descriptors also manage instances of objects that can be re-used by the system, avoiding the need to create a new instance of an object (such as an operation) each time it is used.

Cases in which an iTool developer will need to know about or use object descriptors rather than object identifiers are very rare. We mention object descriptors here because they are used extensively in the iTool object hierarchy to expose the functionality of objects that are created as needed, rather than being created automatically when the iTool is created.

# iTool Object Hierarchy

The iTool system is a collection of object class instances organized in a hierarchy of container objects. The hierarchy serves both to organize the numerous object instances and to display information about the objects in the iTool user interface. In most cases, an object's location in the iTool hierarchy controls where and how the object is made visible in the user interface.

For example, the Rotate operation object is stored in the iImage iTool's object hierarchy with the object identifier

```
OPERATIONS/OPERATIONS/ROTATE
```

From this identifier we can deduce two things:

1. The Rotate operation object is stored in the iTool's object hierarchy in the OPERATIONS container within the OPERATIONS container.
2. The Rotate operation will be displayed in the iTool's widget interface under the **Operations** menu.

## iTool System Object

The *iTool system object* contains and provides a single point of access to all objects managed by the iTool system. Only one instance of the iTool system object can exist in a given IDL session; it is created automatically when any iTool is created.

---

### Note

As an iTool developer, there is no need for you to create or otherwise interact with the system object yourself. This discussion of the structure of the system object is included solely to help you understand the organization of iTool objects.

---

The iTool system object is a subclass of the IDLitContainer object, which provides functionality to manage a hierarchy of container objects via their object identifiers.

# Registering Components

*Registering* an object class links the file containing the IDL code that defines the object (an iTool, a visualization type, an operation, *etc.*) with the object identifier. Objects can be registered either with the iTool system object (in which case their identifiers are fully qualified) or with an individual iTool class (in which case their identifiers are relative to the iTool or to a specific container within the tool).

When an object is registered, it is not immediately instantiated. Instead, the information required to create the object is saved in an object descriptor and placed in the appropriate location in the iTool hierarchy. Later, when the functionality contained in the object is needed, the object descriptor either instantiates the object or provides a reference to an existing instance of the object.

## Registration Methods

Objects are registered using the ITREGISTER procedure (to register the object with the iTool system object) or by calling a Register method on an individual iTool component object.

### Registering Objects with the System Object

Individual iTools, visualization types, and user interface types can be registered with the iTool system object. Of these:

- individual iTools *must* be registered with the system object before they can be created and displayed.
- visualization types *may* be registered with the system object, but can also be registered with an iTool. Visualization types that are registered with the system object will be available to all iTools via the **Insert Visualization** dialog.
- user interface types *must* be registered with the system object; however, creation of new user interfaces is a rare and complex occurrence.

To register an object with the iTool system object, use the ITREGISTER procedure. See “[ITREGISTER](#)” in the *IDL Reference Guide* manual for details and “[Registering a New Tool Class](#)” on page 78 for an example using ITREGISTER.

### Registering Objects with an iTool

Visualization types, operations, manipulators, file readers, and file writers can be registered with an individual iTool. Of these, all must be registered with an individual

iTool except for visualization types, which may have been registered with the iTool system object.

---

**Note**

Many operations, manipulators, file readers, and file writers are registered by the IDLitToolbase class. If you create a new iTool based on this class, these features will be registered automatically. See [“Subclassing from the IDLitToolbase Class”](#) on page 75 for details.

---

**Tip**

If you want some, but not all, of the functionality exposed by the IDLitToolbase class, you may find it useful to subclass from IDLitToolbase and *unregister* one or more features. See the sections on unregistering items in the chapters devoted to creating operations, file readers, and file writers.

---

To register an object with an individual iTool, use one of the Register methods of the IDLitTool class. Register methods exist for each type of object that can be registered (IDLitTool::RegisterOperation for operations, for example). A call to a registration method looks something like this

```
self -> RegisterObject, ObjectName, Object_Class_Name
```

where *Object* is one of the object types that can be registered (Visualization, Operation, Manipulator, FileReader, or FileWriter), *ObjectName* is the string you will use when referring to the object, and *Object\_Class\_Name* is a string that specifies the name of the class file that contains the object’s definition.

See the Register methods under [“IDLitTool”](#) in the *IDL Reference Guide* manual for additional details, and [“Registering a Visualization Type”](#) on page 110, [“Registering an Operation”](#) on page 153, [“Registering a File Reader”](#) on page 177, and [“Registering a File Writer”](#) on page 201 for examples.

## Specifying Object Identifiers

You can use the IDENTIFIER keyword to any of the Register methods to specify an object identifier for the registered object, and thus specify the object’s location in the iTool object hierarchy and in the user interface. If you do not specify a value for the IDENTIFIER keyword, a suitable object identifier will be constructed based on the type of object being registered and the specified *ObjectName*.

## Proxy Registration

You can also register an object as a *proxy* (or *alias*) to another object that has already been registered. Registering an object as a proxy places the proxy object in the iTool

hierarchy in the specified place, but actually calls the original object when a user requests the proxied object. To register a proxy object, specify an object identifier string as the value of the `PROXY` keyword to the `Register` method. For example, the following call to the `RegisterOperation` method places a proxy to the `Undo` object stored in the `iTool` hierarchy under `OPERATIONS/EDIT/UNDO` in the hierarchy under `TOOLBAR/EDIT/UNDO`:

```
self -> RegisterOperation, 'Undo', PROXY = 'Operations/Edit/Undo', $  
    IDENTIFIER = 'Toolbar/Edit/Undo'
```



# iTool Messaging System

*Notifications* are messages sent from one iTool component to one or more *observer* components. The iTool messaging system provides a unified way for components to notify each other of important changes; it is quite general, and can be used to send messages related to any type of change. Some examples:

- Visualizations send notifications when components of the visualization are selected or unselected.
- Notifications are issued when the user changes the value of a property. All visualizations or operations that depend on the value of that property are automatically notified.

---

## Note

Messaging functionality is provided mainly by the [IDLitTool](#) and [IDLitUI](#) objects, using the interface defined by the [IDLitMessaging](#) object.

---

In many cases, the iTool messaging system is transparent to you as an iTool developer; you may never need to create code that uses the messaging system. The main exception to this rule is the creation of user interface panels (discussed in [Chapter 13, “Creating a User Interface Panel”](#)), but there may be other instances in which the notifications sent by the iTool framework itself do not meet your needs and must be augmented by your own message generation and handling code.

## Sending Notifications

To send a notification, an iTool component calls the [IDLitMessaging::DoOnNotify](#) method, providing the object identifier of the component that is sending the notification, a string that uniquely identifies the message being sent, and any value associated with the message. The method call looks like:

```
Obj -> DoOnNotify, IdOriginator, IdMessage, Value
```

where *Obj* is the object calling the DoOnNotify method, *IdOriginator* is the iTool component object identifier string of the component that changed, *IdMessage* is a string that uniquely identifies the change, and *Value* is the value associated with *IdMessage*.

The DoOnNotify method is available to most iTool components, since all components subclass from the IDLitMessaging class either directly or indirectly.

See “[IDLitMessaging::DoOnNotify](#)” in the *IDL Reference Guide* manual for details.

The *IdOriginator* argument is generally the object identifier of an iTool component object, but it can be any string value.

## Notification Messages

The value of the *IdMessage* argument to the *DoOnNotify* method is a string value that must uniquely identify the message being sent. iTool components and callback routines that process notification messages use the value of the *IdMessage* string to determine what action to take when a message arrives from an observed component.

When you call the *DoOnNotify* method yourself, use caution in choosing the value of the *IdMessage* string. If the string you choose conflicts with a message being sent by another iTool component, the message-handling routines may be activated at the wrong time.

## Standard iTool Messages

The following is a list of notification messages sent by components that are part of the standard iTool distribution:

Message String	Meaning
SELECTED UNSELECTED	The selection state of an item being watched had changed. <i>Value</i> contains the object identifier of the component whose selection changed.
SELECTIONCHANGED	The selected item within the current iTool changed. <i>Value</i> contains an empty string.
ADDITEMS MOVEITEMS REMOVEITEMS	A call to the Add, Move, or Remove method of an <i>IDLitContainer</i> that supports the <i>IDLitIMessaging</i> interface is made. <i>Value</i> contains the object identifier of the item that was added, moved, or removed.
SETPROPERTY	The value of a property has been changed on a component. In some cases, <i>Value</i> contains the identifier of the property that changed.
SENSITIVE UNSENSITIVE	The SENSITIVE property of a component has changed. <i>Value</i> contains an empty string.

Table 2-1: Standard iTool Messages.

## Observers

To watch for notifications from an iTool component, an iTool component calls the [IDLitMessaging::AddOnNotifyObserver](#) method, providing the object identifier of the component that is watching and the object identifier of the object being watched as arguments. The method call looks like:

```
Obj -> AddOnNotifyObserver, IdObserver, IdSubject
```

where *Obj* is the object calling the `AddOnNotifyObserver` method, *IdObserver* is the iTool component object identifier string of the component that is watching for notification messages, and *IdSubject* is a string value identifying the item that *IdObserver* is interested in. This is normally the object identifier of an iTool component object, but it can be any string value.

### Note

---

When writing a user interface panel, the *IdObserver* argument contains the object identifier of a user interface adaptor created by a call to the `RegisterWidget` method of the `IDLitUI` class. See [“Creating a UI Panel Interface”](#) on page 243 for details.

---

# System Resources

This section contains information on resources used by the iTool system.

## Icon Bitmaps

Some iTool components have associated icons. Icons for iTool components are displayed in the tree view of a browser window.

Bitmaps used as icons in the iTool system must be either `.bmp` or `.png` files. The images contained in icon bitmap files can be either True Color (24-bit color) images or paletted (8-bit color) images.

### Note

---

There are different requirements for bitmap images that will be displayed on button widgets. See [“Using Button Widgets”](#) in Chapter 27 of the *Building IDL Applications* manual for details.

---

By default, bitmap files for icons used by the iTool system are stored in the `bitmaps` subdirectory of the `resource` subdirectory of the IDL distribution. If an icon’s bitmap file is located in this directory, specify the base name of the file — *without the filename extension* — as the value of the `ICON` property of the component. For example, to use the file `arrow.bmp`, located in the `resource/bitmaps` subdirectory of the IDL distribution, specify the value of the `ICON` property as follows:

```
ICON = 'arrow'
```

If you include the filename extension when setting the `ICON` property, the iTool system assumes that the specified value is the full path to the bitmap file. For example, to use the file `my_icon.png`, stored in the directory `/home/mydir` as an icon, specify the value of the `ICON` property as follows:

```
ICON = '/home/mydir/my_icon.png'
```

If you are distributing your iTool code to others, you may want to specify a path relative to the location of your code for the icon bitmap files. To retrieve the path to the file containing code for a given routine, you could use code similar to the following:

```
; Use my own Icon bitmap
iconName = 'my_icon.png'
routineName = 'myVisualizationType__define'
routineInfo = ROUTINE_INFO(routineName, /SOURCE)
path = FILE_DIRNAME(routineInfo.path, /MARK_DIRECTORY)
```

```
iconPath = path + iconName
```

This code uses the `ROUTINE_INFO` function to retrieve the path to the file specified by the string `routineName`. It then extracts the directory that contains the file using the `FILE_DIRNAME` function, and concatenates the directory name with the name of the bitmap file contained in the string `iconName`.

---

### Note

The routine specified by `routineName` must have been compiled for the `ROUTINE_INFO` function to return the correct value.

---

Including this code in a routine and setting the `ICON` property equal to the variable `iconPath` provides a platform-independent method for locating bitmap files in a directory relative to the directory from which your iTool code was compiled.

If the value of the `ICON` property is not set and the iTool system needs to display an bitmap to represent a component, the file `resource/bitmaps/new.bmp` is used.

## Help System

The iTool system allows the user to select “Help on Selected Item” from the Help menu (or, in the case of the Operations and System Preferences browsers, from the context menu) to display online help for the selected item.

---

### Note

Help for iTool items is provided via a call to the IDL `ONLINE_HELP` procedure. It is beyond the scope of this chapter to discuss the creation of help files suitable for display by `ONLINE_HELP`; please see [Chapter 19, “Providing Online Help For Your Application”](#) in the *Building IDL Applications* manual for additional information.

---

Information about the topic to be displayed by `ONLINE_HELP` is contained in an XML format file named `idliithelp.xml`, located in the `lib/itools/help` subdirectory of the IDL distribution.

The format for a help entry is:

```
<Topic>
  <Keyword>helpKeyword</Keyword>
  <Link type="MSHTMLHELP">contextNumber</Link>
  <Link type="PDF" book=bookName>pdfDestination</Link>
  <Link type="HTML">htmlFile</Link>
</Topic>
```

Where:

- *helpKeyword* is the iTool object class name of the selected object. There can be multiple `<Keyword>` entities for a given `<Topic>`, but they must all precede any `<Link>` entities.
- *contextNumber* is an integer used by the Microsoft Windows HTMLHelp viewer to select a topic from the specified `.chm` or `.hlp` file.
- *pdfDestination* is a string used by the Adobe Acrobat Reader software to select a topic from the specified `.pdf` file.
- *htmlFile* is a string that specifies the name of an HTML file to display in the default browser.
- *bookName* is an optional attribute that specifies the name of the file that contains the HTMLHelp *contextNumber* or the *pdfDestination* specified as the value of the `<Link>` entity.

The `type` attribute of the `<Link>` entity is required, and can have one of the following values:

- MSHTMLHELP
- PDF
- HTML

If more than one `<Link>` entry is present, IDL will choose which to display based on the platform; on Windows platforms, the `<Link>` entity with the `type` attribute set to `MSHTMLHELP` will be used, on Unix platforms, the `<Link>` entity with the `type` attribute set to `PDF` will be used. If the appropriate platform-specific `<Link>` is not present, the first `<Link>` entity of a type that can be displayed on the current platform will be used.



# Chapter 3: Data Management

This chapter describes the iTool data management system.

---

Overview .....	32	Predefined iTool Data Classes .....	38
iTool Data Manager .....	33	Parameters .....	41
iTool Data Types .....	34	Data Type Matching .....	43
iTool Data Objects .....	36	Data Update Mechanism .....	45

# Overview

The iTools system is designed to turn raw data — numbers stored in computer memory — into visualizations that convey information to the viewer. Using data to create a visual display requires some way to route each piece of data to the appropriate part of the algorithm that displays it. In the terminology used by the iTool system, each data item must be associated with a *parameter* of a *visualization*.

The iTools system manages the relationship between data and the visualizations that display data via two mechanisms: *iTool data types* and *parameter data types*. The iTool data type is a property of an IDLitData object (or of an object that inherits from the IDLitData object); it can be any valid scalar string. iTool data types are described in detail in “[iTool Data Types](#)” on page 34. Parameter data types are assigned when a visualization object registers its parameters with the iTool system; they also can be any valid scalar string. Parameter data types are described in “[Parameters](#)” on page 41.

---

**Note**

iTool operations, which do not support the concept of parameters or parameter names, determine whether they can act on a given data object solely on the basis of the iTool data type.

---

The iTool data type and parameter data types are used to match up data objects with visualizations that need data to display. See “[Data Type Matching](#)” on page 43 for a description of how matches are made.

This chapter describes data-management tasks undertaken by the iTool developer. Interactive users manipulate data using a graphical interface known as the iTool Data Manager; this interface allows the user to select and import data items into the iTool system and to manually associate data items with parameters. See [Chapter 2](#), “[Importing and Exporting Data](#)” in the *iTool User’s Guide* manual for a complete description of the Data Manager and its use.



# iTool Data Manager

Data imported into the iTool system is stored in a separate data object hierarchy that is available to all iTools. When a data item is placed in the data manager hierarchy, whether interactively by a user or automatically by some operation of an iTool, the data item is immediately visible to all iTools. The hierarchy of the data manager reflects the hierarchy of the data containers (IDLitDataContainer and IDLitParameterSet objects) it holds.

Unless you are creating new data items within an iTool operation, it is unlikely that you will need to add data to (or remove data from) the data manager yourself. Addition of data items to the data manager is handled automatically if data is imported via any of the standard iTool data import mechanisms (choosing **Open** from the **File** menu, or clicking an **Import** button in the Data Manager user interface).

## Adding Data to the Data Manager

To add an IDLitData, IDLitDataContainer, or IDLitParameterSet object to the data manager, call the [IDLitContainer::AddByIdentifier](#) method on your iTool object with the identifier string '/Data Manager' (note that identifier strings can include spaces, as between the words “Data” and “Manager”):

```
; Create an IDLitDataObject
oData = OBJ_NEW('IDLitData', myData, IDENTIFIER = 'Cool Data')

; Get a reference to the current iTool object.
; (The GetTool method is inherited from the IDLitIMessaging
; class.)
oTool = self -> GetTool()

; Add the data object to the data manager
oTool -> AddByIdentifier, '/Data Manager', oData
```

This results in the oData data object being stored in the data manager with the identifier '/Data Manager/Cool Data'.

See “[iTool Object Identifiers](#)” on page 19 for additional information on identifier strings.

## Removing Data from the Data Manager

To remove data from the data manager, call the [IDLitContainer::RemoveByIdentifier](#) method on your iTool object with the full identifier string used to add the data object:

```
oTool -> RemoveByIdentifier, '/Data Manager/Cool Data'
```

# iTool Data Types

Every iTool data item (IDLitData object or IDLitDataContainer object) has an associated *iTool data type*. The iTool data type of a data item is specified via the TYPE property of the data object, which can contain any scalar string.

---

**Note**

Do not confuse iTool data types with IDL's inherent data types — integers and floating-point integers of various sizes and precisions, strings, structures, pointers, and object references. iTool data types are used only by the iTool system when matching data objects with the parameters expected by a visualization or operation. IDL data types describe how a value or values are stored in computer memory. iTool data types need not correspond directly to an IDL data type.

---

iTool data typing allows the iTool system to match up data objects with visualization parameters even if the data objects have not been explicitly associated with the visualization parameters. Similarly, an iTool operation may apply only to specific forms of data; the iTool data typing mechanism allows an operation to “see” only data of the appropriate type.

## Composite Data Types

Because IDLitData objects can be collected in IDLitDataContainer objects (and, by extension, IDLitParameterSet objects), it is possible that data objects with different iTool data types will be collected in a single container. The iTool data typing system allows these heterogeneous data sets to be named with unique iTool data types that reflect the contents of the container. For example, you might define a data container that contains IDLitData objects with the iTool data types of IDLVECTOR and IDLARRAY2D with your own iTool data type, such as MY\_PLOT.

## Data Types of iTool Components

Since the iTool data type of a data item can be any scalar string value, it is up to the iTool developer to ensure that a data object assigned a given iTool data type contains the data expected by visualizations and operations that accept that type.

Visualizations or operations that accept an iTool data type are written to act on data items that have specific IDL data types (or collections of specific IDL data types, in the case of compound data types). If the data object contains data in a format not expected by the visualization or operation, errors or unexpected behaviors may result.

**Table 3-1** lists the iTool data types defined by the standard iTools included with IDL. You should avoid using these iTool data type names when defining data objects that do not match the contents listed here; if data objects with different contents are given these iTool data type names, portions of the standard iTool functionality may no longer function correctly.

iTool Data Type	Contents
IDLARRAY2D	A two-dimensional array of any IDL data type.
IDLARRAY3D	A three-dimensional array of any IDL data type.
IDLIMAGE	A composite data type that includes IDLIMAGEPIXELS and IDLPALETTE data.
IDLIMAGEPIXELS	One or more two-dimensional image planes.
IDL_OPACITY_TABLE	A 256-element byte array
IDLPALETTE	A 3 x 256-element byte array
IDLPOLYVERTEX	A composite data type that contains a vector of vertex data and a vector of connectivity data.
IDLVECTOR	A vector of any IDL data type.
IDLVERTEX	A vector containing vertex data.

*Table 3-1: iTool data types used by the standard iTools shipped with IDL.*

In addition to avoiding use of the standard iTool data type names for new data types, you should consider using unique naming schemes for iTool data types you create. Choosing your own iTool data type naming scheme will help to avoid conflicts with iTools built by others. This is especially important if you intend to share your iTool code with other IDL users. Choosing a unique prefix or suffix for your iTool data type names should guard against most namespace collisions.

# iTool Data Objects

Each item of data used by an iTool must be encapsulated in an IDLitData object. Data objects can be grouped into collections using the IDLitDataContainer class or its subclass, IDLitParameterSet.

## Data Objects

IDLitData objects can hold data items of any IDL data type. The IDLitData class provides iTool data typing and data change notification functionality, and when coupled with the IDLitDataContainer object forms the base element for the construction of composite data types.

IDLitData objects implement the iTools *notifier interface*, which provides a mechanism by which observers of a data item can be alerted when the state of the information contained in the data object changes. See “[Data Update Mechanism](#)” on page 45 for details on the notification system.

Data objects are created using standard IDL object-creation syntax. For example, to create a data object that contains a vector of data:

```
; Create a data vector containing 10 random values
myData = RANDOMU(seed, 10)
; Create a new data object from the vector.
oData = OBJ_NEW('IDLitDataIDLVector', myData)
```

The IDLitDataIDLVector class is a subclass of IDLitData designed to hold vector data. See “[IDLitData](#)” in the *IDL Reference Guide* manual for a complete description of the data object, its methods, and its properties.

## Data Containers

IDLitDataContainer objects can hold any number of IDLitData or IDLitDataContainer objects. This ability to organize data into object hierarchies allows for the creation of composite data types.

Data container objects are created using standard IDL object-creation syntax, and individual data objects are included in the data container via a call to the IDLitContainer::Add method. For example, the following statements create a new data container and add the data object created in the previous section:

```
; Create a data container
oDataContainer = OBJ_NEW('IDLitDataContainer')
; Add a data object.
oDataContainer -> Add, oData
```

In this example we do not specify an iTool data type for the data container object itself.

---

**Tip**

Often, you will organize data using a subclass of the IDLitDataContainer class: the IDLitParameterSet.

---

See “[IDLitDataContainer](#)” in the *IDL Reference Guide* manual for a complete description of the data container object, its methods, and its properties.

## Parameter Sets

The IDLitParameterSet class is a specialized subclass of the IDLitDataContainer class that provides the ability to associate parameters with the contained IDLitData and IDLitDataContainer objects. This association allows the iTool developer to package a set of data parameters in a single container, which is then provided to the iTools system for processing and display. See “[IDLitParameterSet](#)” in the *IDL Reference Guide* manual for a complete description of the parameter set object, its methods, and its properties.

---

**Note**

Do not confuse *parameter sets*, which are containers for data objects, with *parameters*, which define how data is used by a visualization object. Parameters are described in “[Parameters](#)” on page 41.

---

Using a parameter set object is very similar to using a data container object. The parameter set itself is created using standard IDL object-creation syntax. The parameter set object allows for the association of a parameter with each added data object. For example, the following statements create a new parameter set and add the data object created in the previous section, assigning a parameter:

```
; Create a parameter set object
oParameterSet = OBJ_NEW('IDLitParameterSet')
; Add a data object, assigning a parameter
oParameterSet -> Add, oData, PARAMETER_NAME = 'Y data'
```

# Predefined iTool Data Classes

The iTool system distributed with IDL includes a number of predefined data classes. The predefined classes are subclasses of the `IDLitData` class; each performs initialization steps that are commonly used when creating data objects that contain data of specific composite data types. Some of the predefined data classes create data sub-containers to hold associated data objects, and some register properties associated with the data.

## Note

---

The predefined iTool data subclasses are provided as a convenience. You can always create a generic `IDLitData` object rather than using one of the predefined classes.

---

You can create objects of these data classes in the same way you create a generic data object: by calling the `OBJ_NEW` function and specifying the appropriate class name. You can also create new specialized data classes based on one of the predefined classes. Data classes are located in the `lib/itools/components` subdirectory of the IDL directory.

## IDLitDataIDLArray2D

Creates an `IDLitData` object of whose `TYPE` property is set to `IDLARRAY2D`. Used to store a two-dimensional array of any IDL data type.

### Registered Properties

- None

### Data Sub-containers

- None

## IDLitDataIDLArray3D

Creates an `IDLitData` object of whose `TYPE` property is set to `IDLARRAY3D`. Used to store a three-dimensional array of any IDL data type.

### Registered Properties

- None

### Data Sub-containers

- None

## IDLitDataIDLImage

Creates an IDLitData object of whose TYPE property is set to IDLIMAGE. Used to store two-dimensional image data. Images can be constructed from multiple image planes.

### Registered Properties

- INTERLEAVE

### Data Sub-containers

- An IDLitDataIDLPalette object named “Palette” that contains palette information provided as an argument to the Init method.
- An IDLitDataIDLImagePixels object named “Image Planes” that contains the image data provided as an argument to the Init method.

## IDLitDataIDLImagePixels

Creates an IDLitData object of whose TYPE property is set to IDLIMAGEPIXELS. Used to store the raw image data (pixels).

### Registered Properties

- INTERLEAVE

### Data Sub-containers

- None

## IDLitDataIDLPalette

Creates an IDLitData object of whose TYPE property is set to IDLPALETTE. Used to store palette data.

### Registered Properties

- None

### Data Sub-containers

- None

## IDLitDataIDLPolyvertex

Creates an IDLitData object of whose TYPE property is set to IDLPOLYVERTEX. Used to store vertex and connectivity lists suitable for use with the IDLgrPolygon and IDLgrPolyline objects.

**Registered Properties**

- None

**Data Sub-containers**

- An IDLitData object named “Vertices” that contains the vertex list.
- An IDLitData object named “Connectivity” that contains the connectivity list.

**IDLitDataIDLVector**

Creates an IDLitData object of whose TYPE property is set to IDLVECTOR. Used to store a one-dimensional array of any IDL data type.

**Registered Properties**

- None

**Data Sub-containers**

- None



# Parameters

*Parameters* represent data items used in a well-defined way by an algorithm that is computing a result. In the scheme of the iTools, parameters are the raw material fed to *visualization objects* — the IDL routines that create visual displays.

For example, a visualization object that creates a simple line plot might require two parameters: vectors of dependent and independent data values. These two vectors would be passed to the routines within the visualization object for processing, and the result would be displayed in the iTool window.

When a visualization object is created, it *registers* one or more parameters with the iTool system. Each parameter has a *parameter name* and can be of one or more *iTool data types*. Parameter names are used to route the individual data items to the correct routines within the visualization object. See [Chapter 6, “Creating a Visualization”](#) for more on creating visualization objects.

## Note

---

Do not confuse *parameters*, which define how data is used by a visualization object, with *parameter sets*, which are containers for data objects. Parameter sets are described in [“Parameter Sets”](#) on page 37.

---

## Parameter Names

Each parameter registered by a visualization is given a parameter name. The parameter name is a scalar string, and its scope is the visualization by which it is registered. Different visualizations can register parameters that have different properties using the same parameter name.

## Parameter Data Types

Each parameter registered by a visualization is associated with one or more iTool data types by setting the TYPES property. The value of the TYPES property can be a scalar string or a string array; a single parameter can be associated with multiple data types. See [“iTool Data Types”](#) on page 34 for more on iTool data types.

## Registering Parameters

Parameters are *registered* when a visualization is created; that is, in the Init method of an iTool visualization class. To register a parameter, call the RegisterParameter

method of the `IDLitParameter` class (of which `iTool` visualization classes are a subclass):

```
self -> RegisterParameter, ParmameterName, $
      TYPES = ['DataType1', ..., 'DataTypeN']
```

where *ParameterName* is a string that defines the name of the parameter and the `TYPES` keyword is set equal to a string or array of strings specifying the `iTool` system data types the parameter can represent. (See “[iTool Data Types](#)” on page 34 for information on `iTools` data types.)

A typical parameter registration call looks like the following:

```
self->RegisterParameter, 'Y', /INPUT, TYPES='IDLVECTOR', /OPTARGET
```

Here, the string argument `Y` is the name of the parameter being registered. The `INPUT` keyword specifies that `Y` is an input parameter (specified by the method’s caller), the `TYPES` keyword specifies that `Y` is a vector, and the `OPTARGET` keyword specifies that operations can be performed on the `Y` vector.

Additional keywords can be set in the call to `RegisterParameter`. See the documentation for “[IDLitParameter::RegisterParameter](#)” in the *IDL Reference Guide* manual for additional details.

# Data Type Matching

To understand how the iTool data type matching system works, consider the following:

- When a visualization is created, it registers one or more parameters, assigning a *parameter name* and one or more *iTool data types* to each.
- When a data object is imported or created by an iTool, it is assigned one or more iTool data types.
- When a parameter set object is created to contain data objects, each data object can optionally be assigned one or more *parameter names*.

Now assume that an iTool user requests that a particular visualization be created from a particular collection of data objects, which are stored in a parameter set object. The iTool system will do the following:

1. Retrieve the parameter name and iTool data types registered for the visualization's first parameter.
2. If the parameter set object contains a data object whose Parameter Name matches the parameter name of the visualization's first registered parameter, use that data object as the data for the visualization parameter.
3. If the parameter set object does *not* contain a data object with a matching Parameter Name, check the parameter set for data objects for which the Parameter Name property is not set. If there are no data objects without Parameter Names, no data is associated with the visualization parameter.
4. Check the iTool data types of the data objects without Parameter Names. If a data object whose iTool data type matches the list of registered data types for the visualization parameter is found, use that data object as the data for the visualization parameter. If no data objects match any data types, no data is associated with the visualization parameter.
5. Repeat until all registered visualization parameters have been either populated with data, skipped, or there are no more data objects to supply data.

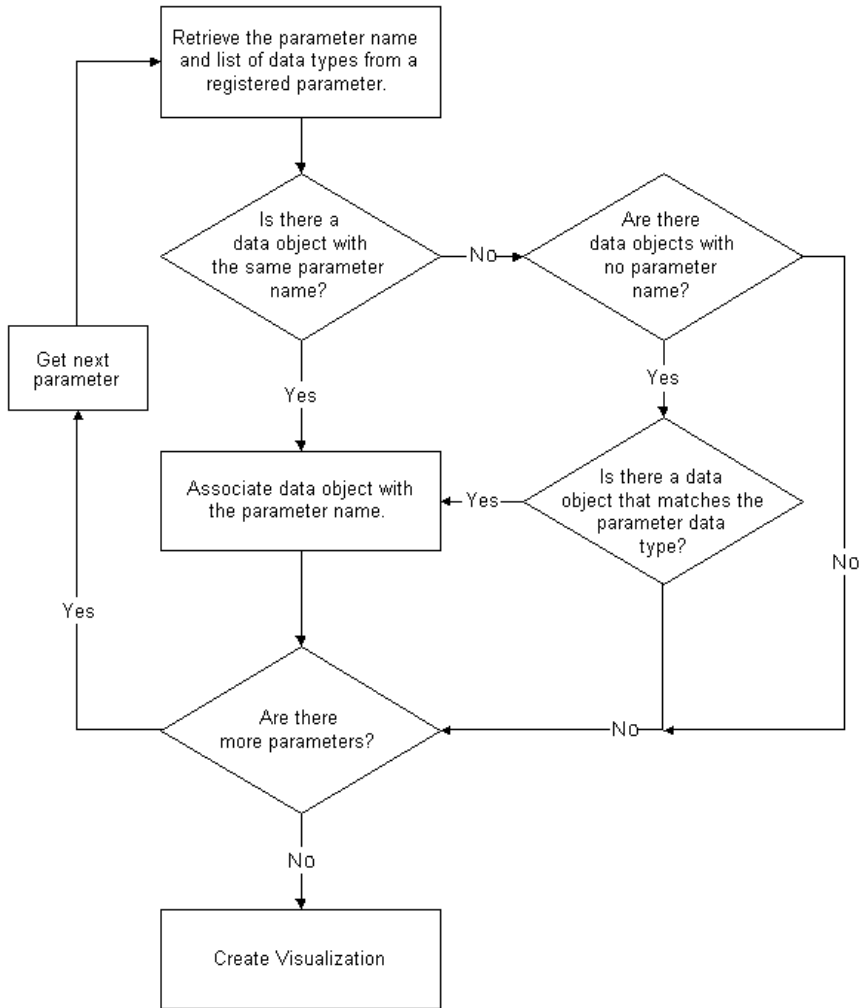
---

**Note**

Parameter name matching is done in a case-insensitive fashion. If a parameter is registered with the parameter name “MyParameter” and a data object has its Parameter Name property set to “myParameter”, the two will match.

---

The [Figure 3-1](#) illustrates this process as a flow diagram.



*Figure 3-1: Data type matching algorithm used by iTools.*

# Data Update Mechanism

When the data contained in a data item changes (usually as the result of the application of a data-centric operation), all visualizations that depend on that data item are automatically notified of the change via a call to the visualization object's `OnDataChangeUpdate` method. (See [“Creating an OnDataChangeUpdate Method”](#) on page 105 for details.)

The data update mechanism is automatic; if you have assigned iTTool data types (and, optionally, parameter names) to your data objects, the data matching mechanisms of the `IDLitParameter` interface will ensure that updates happen when necessary. Unless you have modified core iTTool functionality, you do not need to handle data change updates yourself.





# Chapter 4: Property Management

This chapter describes the iTool property interface.

---

About the Properties Interface .....	48	Property Attributes .....	58
Property Data Types .....	51	Property Aggregation .....	61
Registering Properties .....	54	Property Update Mechanism .....	63
Property Identifiers .....	57	Properties of the iTools System .....	64

# About the Properties Interface

Object *properties* are used to store settings and values that relate to visualizations, data, and other components of an iTool. The iTools system presents a graphical *property sheet* interface to tool users; see “[Property Sheets](#)” in Chapter 3 of the *iTool User’s Guide* manual for a description of the property sheet interface. As a tool developer, you can manage individual property values, as well as the property set that is visible to users of your application, programmatically.

---

**Note**

In most cases, you do not need to manage updates to visualizations or data that result from a user’s modifications to values in a property sheet. See “[Property Update Mechanism](#)” on page 63 for details.

---

## What is a Property?

A property is a value that is associated with an object instance. Examples of property values commonly associated with iTool objects are Boolean True/False flags, text strings, color values stored as RGB triplets, and integer and floating point values. For example, a plot visualization object might have a **Color** property that defines the line color as an RGB triplet, a **Line thickness** property that defines the thickness of the line drawn as an integer value in pixels, and a **Name** property that defines how the plot is referred to in iTool browser windows.

## Properties vs. Preferences

In the case of objects that have a visual representation (plots, annotations, surfaces, axes, *etc.*), properties apply to a single instance of an object. When a new instance of the same type of object is created, any property changes applied to the first object are not applied to the second. For example, if you change the color of a plot line to red, subsequent plot lines will still be created with the default line color.

In the case of non-visual objects (operations, file readers and writers, and manipulators) only one instance of the object is created no matter how many times the object is requested. As a result, properties set on these objects will “stick” until changed again. For example, if you change the value of the Width property of the Smooth operation, the property will retain the value you set until you change it again or close that iTool.

Finally, properties that apply to all iTools and which are preserved between iTool sessions are known as *preferences*. Preferences include default values for properties



of visual objects (default line style, colors, *etc.*), and default properties for file readers, and file writers.

## How are Properties Displayed?

Any iTool object can have properties. Properties are always displayed via the iTool property sheet interface, which uses the IDL `WIDGET_PROPERTYSHEET` function to present property names and values in a columnar display. The way the property sheet interface is displayed to iTool users depends on the type of object for which properties are being displayed.

- For visualization objects (any graphical item that appears in the iTool window), the property sheet can be displayed by double-clicking on an item in the iTool window, by selecting Properties from the window context menu, or by selecting **Visualization Browser** from the **Window** menu.
- For operations, the property sheet can be displayed by selecting **Operations Browser** from the **Operations** menu.
- For system preferences, the property sheet can be displayed by selecting **Preferences** from the **File** menu.

## Setting and Retrieving Property Values

iTool property values are set and retrieved like all object property values, via `SetProperty` and `GetProperty` methods. See “[IDLitComponent::SetProperty](#)” and “[IDLitComponent::GetProperty](#)” in the *IDL Reference Guide* manual for details, but remember that your own object classes will be responsible for implementing these methods and handling the actual property values. See the chapters in “[Using the iTools Component Framework](#)” for examples of `GetProperty` and `SetProperty` methods.

## Property Data Types

While object properties can contain any value that can be stored in IDL, the iTool property sheet interface (based on the `WIDGET_PROPERTYSHEET` routine) will only display properties of nine pre-defined property data types. (See “[Property Data Types](#)” on page 51 for descriptions of the pre-defined types.) In addition, the property sheet interface allows developers to build and associate a separate widget-based user interface that allows iTool users to specify data values of any IDL data type. User-defined property values are discussed in “[User Defined Property Types](#)” on page 53.

## Property Registration

In order for an object property to be displayed by the graphical property sheet interface, it must be registered with the iTool system. Properties are generally registered when an object is created; see [“Registering Properties”](#) on page 54 for additional details.

## Property Identifiers

Properties are referenced within the iTools system using property identifiers, which are simple scalar strings defined when the property is registered. See [“Property Identifiers”](#) on page 57 for details.

## Property Attributes

In addition to the property value, properties have attributes that affect the way the property is displayed in the property sheet user interface. See [“Property Attributes”](#) on page 58 for details.

## Property Aggregation

Visualization objects can be built from any number of atomic IDL graphic objects and iTool visualization objects. The property aggregation mechanism allows the properties of all of the objects in a visualization to be displayed in a single property sheet. See [“Property Aggregation”](#) on page 61 for details.

# Property Data Types

Registered properties must be of one of the data types listed in [Table 4-1](#).

## Note

Properties of objects that are *not* registered (that is, properties that cannot appear in a property sheet) can be of any IDL data type.

Type Code	Type	Description
0	USERDEF	User Defined properties can contain values of any IDL type, but must also include a string value that will be displayed in the property sheet. See <a href="#">“User Defined Property Types”</a> on page 53 for additional discussion of User Defined property types.
1	BOOLEAN	Boolean properties contain either the integer 0 or the integer 1.
2	INTEGER	Integer properties contain an integer value. If a property of integer data type has a <code>VALID_RANGE</code> attribute that includes an increment value, the property is displayed in a property sheet using a slider. If no increment value is supplied, the property sheet allows the user to edit values manually.
3	FLOAT	Float properties contain a double-precision floating-point value. If a property of float data type has a <code>VALID_RANGE</code> attribute that includes an increment value, the property is displayed in a property sheet using a slider. If no increment value is supplied, the property sheet allows the user to edit values manually.
4	STRING	String properties contain a scalar string value
5	COLOR	Color properties contain an RGB color triplet

*Table 4-1: iTools property data types.*

Type Code	Type	Description
6	LINESTYLE	<p>Linestyle properties contain an integer value between 0 and 6, corresponding to the following IDL line styles:</p> <ul style="list-style-type: none"> <li>• 0 = Solid</li> <li>• 1 = Dotted</li> <li>• 2 = Dashed</li> <li>• 3 = Dash Dot</li> <li>• 4 = Dash Dot Dot</li> <li>• 5 = Long Dashes</li> <li>• 6 = No Line</li> </ul> <p>See <a href="#">Appendix B, “Property Controls”</a> in the <i>iTool User’s Guide</i> manual for a visual example of the available line styles.</p>
7	SYMBOL	<p>Symbol properties contain an integer value between 0 and 8, corresponding to the following IDL symbol types:</p> <ul style="list-style-type: none"> <li>• 0 = No symbol</li> <li>• 1 = Plus sign</li> <li>• 2 = Asterisk</li> <li>• 3 = Period (Dot)</li> <li>• 4 = Diamond</li> <li>• 5 = Triangle</li> <li>• 6 = Square</li> <li>• 7 = X</li> <li>• 8 = “Greater-than” Arrow Head (&gt;)</li> <li>• 9 = “Less-than” Arrow Head (&lt;)</li> </ul> <p>See <a href="#">Appendix B, “Property Controls”</a> in the <i>iTool User’s Guide</i> manual for a visual example of the available symbols.</p>

Table 4-1: *iTools* property data types.

Type Code	Type	Description
8	THICKNESS	Thickness properties contain an integer value between 1 and 10, corresponding to the thickness (in points) of the line.
9	ENUMLIST	Enumerated List properties contain an array of string values defined when the property is registered. The <code>GetProperty</code> method returns the zero-based index of the selected item.

Table 4-1: *iTools* property data types.

## User Defined Property Types

The User Defined property type lets you to create a custom interface that allow users of your *iTool* to select data of types other than the predefined *iTool* property types. Creating a user defined property type entails the following:

- Creating a `EditUserDefProperty` method for the *iTool* component (usually a visualization or operation) that uses the user defined property. See [“IDLitComponent::EditUserDefProperty”](#) in the *IDL Reference Guide* manual for details.
- Creating user interface code to allow users to select a value. In the initial release of the *iTool* system, this means writing an IDL widget interface, but in future releases other users interfaces may be available.
- Creating a *user interface service* to display the interface. See [Chapter 12, “Creating a User Interface Service”](#) for details.

# Registering Properties

In order for a property associated with an iTool component to be included in the property sheet for that component, the property must be *registered* with the iTool. The property registration mechanism accomplishes several things:

- It allows you to expose as many or as few of the properties of an underlying object as you choose.
- It allows you to add user-defined properties to existing objects, and expose those new properties to users of your application.

---

## Note

You can write code to access and change property values programmatically, even if the property being changed is not registered.

---

## Registering a Property

Register a property by calling the `RegisterProperty` method of the `IDLitComponent` class:

```
self -> RegisterProperty, PropertyIdentifier [, TypeCode] $
    [, ATTRIBUTE = value]
```

where *PropertyIdentifier* is a string that uniquely identifies the property, *TypeCode* is an integer between 0 and 9 specifying the property data type, and *ATTRIBUTE* is a property attribute. You can specify multiple property attributes in the call to `RegisterProperty`; see “[Property Attributes](#)” on page 58 for details.

---

## Note

The property identifier string must obey certain rules; see “[Property Identifiers](#)” on page 57 for details.

---

You can omit the *TypeCode* parameter and specify a type keyword; the following two method calls are identical:

```
self -> RegisterProperty, 'MYPROPERTY', 1

self -> RegisterProperty, 'MYPROPERTY', /BOOLEAN
```

See “[Property Data Types](#)” on page 51 for a list of property data types, their type codes, and the associated keywords to the `RegisterProperty` method.

A typical property registration call looks like the following:

```
self -> RegisterProperty, 'FONT_STYLE', $
    ENUMLIST = ['Normal', 'Bold'], $
    NAME = 'Font style'
```

Here, the string argument `FONT_STYLE` is the property identifier of the property being registered; this string must be the same as the name of the keyword used with the `GetProperty` or `SetProperty` method when changing the value of the property.

The `ENUMLIST` keyword specifies that the property data type is an enumerated list of strings containing two possible property values (`'Normal'`, `'Bold'`); this will appear as a pulldown list of values in the property sheet. The `NAME` keyword specifies the string that will be used as the label for the property in the property sheet; if `NAME` is omitted, the property identifier string will be used in the property sheet.

---

**Note**

Values set via keywords to the `RegisterProperty` method are known as *property attributes*. Property attributes can be modified after registration using the `SetPropertyAttribute` method, described in [“Property Attributes”](#) on page 58.

---

Additional keywords can be set in the call to `RegisterProperty`. See the documentation for [“IDLitComponent::RegisterProperty”](#) in the *IDL Reference Guide* manual for additional details.

In addition to registering the property using `RegisterProperty`, you must make sure that the `GetProperty` and `SetProperty` methods of your object handle the value of the property being registered.

## Pre-Registered Properties

Not all properties need to be explicitly registered in your iTool code in order to be displayed in a property sheet. Most of the IDL graphics objects (`IDLgrAxis`, `IDLgrPlot`, *etc.*) have a set of properties that are automatically registered if you set the `REGISTER_PROPERTIES` property of the object to 1 when it is instantiated. See the list of object properties contained in the documentation for the IDL graphics objects in the *IDL Reference Guide* to determine which properties are registered when the `REGISTER_PROPERTIES` property is set.

There may be times when you want some, but not all, of the registrable properties of a graphics object to appear in the property sheet interface. You have two options in this case:

1. Register the properties of the graphics object individually, with calls to the `RegisterProperty` method.

2. Use the REGISTER\_PROPERTIES keyword when instantiating the graphics object, then set the HIDE property attribute on the properties you want to remove from the property sheet. See [“Property Attributes”](#) on page 58 for more on this option.



# Property Identifiers

Property *identifiers* are scalar string values that identify a registered property. The property identifier string *must* be accepted as a keyword by the `GetProperty` and `SetProperty` methods for the object. Like all IDL keywords, property identifier strings must be valid IDL variable names, and cannot contain spaces or non-alphanumeric characters other than “\_”, “!”, and “\$”. See “[IDL\\_VALIDNAME](#)” in the *IDL Reference Guide* manual for details on valid IDL variable names.

---

**Note**

You can specify the property identifier string using any case; IDL will match the property identifier with the `GetProperty` or `SetProperty` keyword in a case-insensitive manner. As a matter of style, using upper case letters when specifying property identifiers helps someone reading your code visually match the property identifier with the keyword values.

---

The property identifier is *not* displayed in the property sheet interface; the value of the `NAME` property attribute is displayed instead. However, if you do not supply the `NAME` attribute, the iTool system will assign it the same value as the property identifier.

# Property Attributes

Property *attributes* are values associated with a property that affect the way the property is displayed in the iTool property sheet interface. Attributes could be considered *properties-of-properties*; as with actual properties, special methods are used to get and set attribute values.

## Note

---

A property must be *registered* in order to set or retrieve attribute values.

---

Property attributes can be set in the call to the `IDLitComponent::RegisterProperty` method; simply include the attribute name and its value as a keyword-value pair.

If a property has already been registered, you can change the registered attribute values using the `SetPropertyAttribute` method of the `IDLitComponent` class:

```
self -> SetPropertyAttribute, PropertyIdentifier, ATTRIBUTE = value
```

where *PropertyIdentifier* is a string that uniquely identifies the property, *ATTRIBUTE* is one of the property attributes described in “[Available Property Attributes](#)” on page 58, and *value* is the attribute value. See “[Property Identifiers](#)” on page 57 for a discussion of property identifier strings.

A typical property attribute modification call looks like the following:

```
self -> SetPropertyAttribute, 'COLOR', NAME = 'Surface color'
```

Here, we change the Name attribute of the COLOR property; when this property is displayed in a property sheet, the label will be `Surface color`.

See “[IDLitComponent::SetPropertyAttribute](#)” in the *IDL Reference Guide* manual for additional details.

## Available Property Attributes

Every registered iTool property has the following attributes. Property attributes can be specified as keywords to the `RegisterProperty` method of the `IDLitComponent` class. Attributes whose names are followed by the word “Get” can be retrieved using the `GetPropertyAttribute` method of the `IDLitComponent` class; attributes whose names are followed by the word “Set” can be set using the `SetPropertyAttribute` method.

## DESCRIPTION (Get, Set)

A string value containing a text description of the property. This string is displayed in the property sheet interface.

## ENUMLIST (Get, Set)

An array of string values to be displayed in the property sheet interface as an *enumerated list*. This property type allows the user to select a string value from a dropdown list in the user interface, but returns the integer index of the selected item as the value of the property. This attribute is only used by properties of TYPE = 9 (enumerated list).

## HIDE (Get, Set)

A Boolean flag that specifies whether the property should be displayed in the property sheet interface.

## NAME (Get, Set)

A string value that is displayed as the property name in the property sheet interface. If the NAME attribute is not specified in the call to the RegisterProperty method, this attribute will be set to the property identifier string.

## PROPERTY\_IDENTIFIER (Get)

A string value containing the property identifier. See [“Property Identifiers”](#) on page 57 for details.

## SENSITIVE (Get, Set)

A Boolean flag that specifies whether the property should be editable by the user when displayed in the property sheet interface. Properties with the SENSITIVE attribute set to 0 are displayed, but are dimmed and are not editable.

## TYPE (Get)

The property data type code for the property. See [“Property Data Types”](#) on page 51 for details.

## UNDEFINED (Get, Set)

A Boolean flag that indicates that the property should appear as a blank cell when displayed in the property sheet interface. This is useful in situations where properties

of multiple objects are displayed in the property sheet (either because multiple objects are selected, or because the objects have been grouped).

---

**Note**

The iTool developer is responsible for setting this property attribute back to zero. Use the `SET_DEFINED` field of the `WIDGET_PROPERTY SHEET` event structure to determine when to set the `UNDEFINED` attribute back to zero.

---

**USERDEF (Get, Set)**

A string that represents the value of a user-defined property. See [“User Defined Property Types”](#) on page 53 for details.

**VALID\_RANGE (Get, Set)**

A two- or three-element array of integers or floating-point values. If the `VALID_RANGE` attribute contains a value, the property sheet interface will allow the user to edit the numerical value by dragging a slider control. The first element of the array represents the minimum allowed value, the second element represents the maximum allowed value, and the third element (if present) represents an increment value. If an increment is specified, only values that are integer multiples of the increment, plus the initial value, are allowed. This attribute is only used by properties of `TYPE = 2` or `TYPE = 3` (integer or float).

# Property Aggregation

The iTools *property aggregation* mechanism allows the properties of several different objects held by the same container object to be displayed in the same property sheet automatically. Without property aggregation, you would have to manually register all of the properties of the objects contained in your visualization type object.

Aggregate the properties of contained objects using the `Aggregate` method of the `IDLitVisualization` class:

```
self -> Aggregate, Object_Reference
```

where *Object\_Reference* is a reference to the object whose properties you want aggregated into the visualization object. A typical property aggregation call looks like the following:

```
self._oSymbol = OBJ_NEW('IDLitSymbol', PARENT = self)
self -> Aggregate, self._oSymbol
```

Here, the first line creates an `IDLitSymbol` object and stores it in the `_oSymbol` field of the visualization object's class structure. The second line calls the `Aggregate` method with the object reference to the `IDLitSymbol` object as the argument. After the call to the `Aggregate` method, all registered properties of the `IDLitSymbol` object will be exposed in the property sheet for the visualization itself.

## Note

---

The `IDLitVisualization::Add` method includes an `AGGREGATE` keyword. This keyword is simply a shorthand method of aggregating the properties of an object during the call to the `Add` method, eliminating the need to call the `Aggregate` method separately. The call

```
self -> Add, Object_Reference, /AGGREGATE
```

is the same as the following two calls:

```
self -> Add, Object_Reference
self -> Aggregate, Object_Reference
```

---

## Working with Aggregated Properties

When the properties of multiple objects are aggregated in a visualization object, there are two possible ways to display the combined property set: a *union* or an *intersection*. The way aggregated properties are displayed by a given visualization

depends on the value of the visualization's `PROPERTY_INTERSECTION` property: by default, this property is not set (it contains a value of 0), and the union of the aggregated properties is displayed. If `PROPERTY_INTERSECTION` is set to 1 when the visualization object is created, the intersection of the aggregated properties is displayed. The following sections explain the behavior of the property sheet interface in both situations.

## Union

By default, a visualization object displays the union of the properties of any aggregated objects. Properties are displayed in the property sheet interface as follows:

- All of the unique properties of all of the aggregated objects are displayed.
- Only one instance of a given property is displayed. This means that if multiple objects have the same property, this property will be displayed only once, and all objects will have the same property value.
- The visualization will appear in iTool browsers as a single object — the aggregated objects will not be visible in the browser hierarchy.

## Intersection

If the `PROPERTY_INTERSECTION` property is set when the visualization is created, the visualization object displays the intersection of any aggregated objects. Properties are displayed in the property sheet interface as follows:

- Only properties that are common to all of the aggregated objects are displayed as properties of the visualization object. Changing the value of a common property in the visualization's property sheet changes the value for all aggregated objects.
- The visualization will appear in iTool browsers as a container object — the aggregated objects will be visible beneath the visualization object in the browser hierarchy (unless the property's `HIDE` attribute is set, in which case the property will not be displayed). Selecting an individual aggregated object in the browser hierarchy will display that object's own properties.
- If the value of a property that is common to all of the aggregated objects is different for different objects, the value will show in the parent container's property sheet as undefined.

# Property Update Mechanism

When a user changes the value of a property via the property sheet interface, the object that implements the property is automatically updated. If the object has a visual representation, the display of the iTool window is also updated automatically.

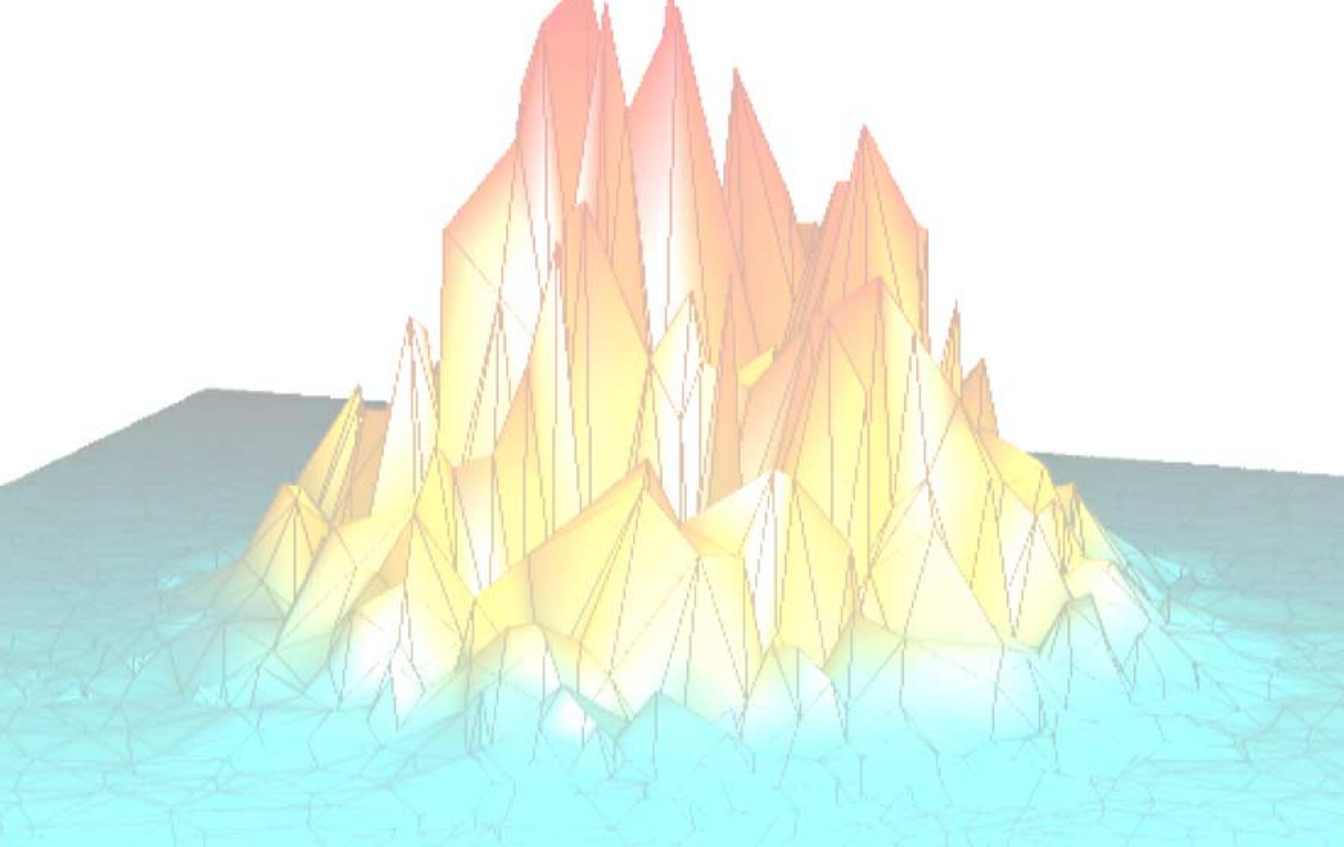
The update mechanism is handled by the `SetProperty` method; as long as any `SetProperty` methods you create call the `SetProperty` methods of their superclasses, there is nothing more you need to do.

Property changes are automatically recorded by the iTool undo/redo system. You do not need to supply any extra code to support undo/redo.

# Properties of the iTools System

iTools *system preferences* are default settings for the values of properties of visualization types, file readers, file writers, and the iTool system itself. System preferences are revealed to the user via the system preferences browser, which is displayed when a user selects **File** → **Preferences** in an iTool





## ***Part II: Using the iTools Component Framework***





# Chapter 5: Creating an iTool

This chapter describes the process of creating a new iTool definition and command-line launch routine.

---

Overview .....	68	Creating an iTool Launch Routine .....	80
Creating a New iTool Class .....	69	Example: Simple iTool .....	85
Registering a New Tool Class .....	78		

# Overview

Creating a new iTool using the iTools component framework is vastly simpler than creating a similar tool from scratch in IDL. The standard iTool user interface and functionality can be included in any new iTool with a few simple lines of code. Using the iTools framework leaves you free to concentrate on developing functionality unique to your application.

That said, creating even the simplest iTool *does* require that you have a basic familiarity with the concepts of object-oriented programming in general, and with the process of creating object-oriented programs in IDL in particular. If you have written even very simple object-oriented applications in IDL, or in another language such as Java or C++, you probably already have the necessary skills. For background information on writing object-oriented applications in IDL, see [Chapter 22, “Object Basics”](#) in the *Building IDL Applications* manual.

## The iTool Creation Process

To create a new iTool, you will do the following:

- Choose an iTool object class on which your new tool will be based. In almost all cases, you will base new iTools either on the IDL*i*Toolbase class or on an iTool class that is itself based on IDL*i*Toolbase. The IDL*i*Toolbase class defines all of the standard iTool functionality exposed by the individual iTools included with IDL.
- Define the visualization types, data operations, user interface tools (manipulators), and data import/export features that will be available in your iTool. You can choose from a variety of predefined features included with the iTool system as included with IDL, or you can create your own. The process of defining the features available in your new iTool is discussed in [“Creating a New iTool Class”](#) on page 69.
- Register your new iTool class with the system as described in [“Registering a New Tool Class”](#) on page 78.
- Provide an IDL procedure that creates an instance of your new iTool class, as described in [“Creating an iTool Launch Routine”](#) on page 80.

This chapter describes the process of creating a new iTool from existing visualization types, operations, and file readers and writers. The chapters that follow describe how to create your own visualization types, operations, and file readers and writers to be incorporated into new iTools.

# Creating a New iTool Class

An iTool object class definition file must contain, at the least, the class Init method and the class structure definition routine. The Init method contains the statements that register any operations, visualizations, and file readers or writers available in the iTool. The class structure definition routine defines an IDL structure that will be used when creating new instances of the iTool object.

The process of creating an iTool definition is outlined in the following sections:

- [“Creating an Init Method”](#) on page 69
- [“Creating the Class Structure Definition”](#) on page 75

## Creating an Init Method

The iTool class Init method handles any initialization required by the iTool object, and should do the following:

- call the Init methods of any superclasses
- register visualizations, operations, and file readers/writers available in the new iTool but not registered by any superclasses
- perform other initialization steps as necessary
- return the value 1 if the initialization steps are successful, or 0 otherwise

## Superclass Initialization

The iTool class Init method should call the Init method of any required superclass. For example, if your iTool is based on an existing iTool, you would call that tool’s Init method:

```
success = self -> SomeToolClass::Init()
```

where *SomeToolClass* is the class definition file for the iTool on which your new iTool is based. The variable *success* contains a 1 if the initialization was successful.

### Note

---

Your iTool class may have multiple superclasses. In general, each superclass’ Init method should be invoked by your class’ Init method.

---

## Error Checking

Rather than simply calling the superclass Init method, it is a good idea to check whether the call to the superclass Init method succeeded. The following statement checks the value returned by the superclass Init method; if the returned value is 0 (indicating failure), the current Init method also immediately returns with a value of 0:

```
IF (self -> SomeToolClass::Init() EQ 0) THEN RETURN, 0
```

This convention is used in all iTool classes included with IDL. RSI strongly suggests that you include similar checks in your own class definition files.

## Keywords to the Init Method

Properties of the iTool class can be set in the Init method by specifying the property names and values as IDL keyword-value pairs. In addition to any keywords implemented directly in the Init method of the superclass on which you base your class, the properties of the IDLITool class are available to any iTool class. See “[IDLITool Properties](#)” in the *IDL Reference Guide* manual.

Use keyword inheritance (the `_EXTRA` keyword) to pass keyword parameters through to the superclass as necessary. See “[Keyword Inheritance](#)” in Chapter 4 of the *Building IDL Applications* manual for details on IDL’s keyword inheritance mechanism.

## Standard Base Class

While you can create your new iTool from any existing iTool class, in many cases, iTool classes you create will be subclassed directly from the base class IDLIToolbase:

```
IF (self -> IDLIToolbase::Init(_EXTRA = _extra) EQ 0) THEN $  
    RETURN, 0
```

The IDLIToolbase class provides the base iTool functionality used in the tools created by RSI. See “[Subclassing from the IDLIToolbase Class](#)” on page 75 for details.

### Note

---

To create an iTool that *does not* include the standard iTool functionality, subclass from the IDLITool class.

---

## Return Value

If all of the routines and methods used in the `Init` method execute successfully, the method should indicate successful initialization by returning 1. Other iTools that subclass from your iTool class may check this return value, as your routine should check the value returned by any superclass `Init` methods called.

## Registering Visualizations

Registering a visualization type with an iTool class allows instances of the iTool to create and display visualizations of that type. Any number of visualization types can be registered for use by a given iTool.

### Note

---

You must register at least one visualization type with your iTool class. Unlike operations, and file readers and writers, no visualization types are registered by the `IDLitToolbase` class.

---

Visualization types are registered by calling the `IDLitTool::RegisterVisualization` method:

```
self -> RegisterVisualization, Visualization_Type, $
      VisType_Class_Name
```

where *Visualization\_Type* is the string you will use when referring to the visualization type, and *VisType\_Class\_Name* is a string that specifies the name of the class file that contains the visualization type's definition.

### Note

---

The file *VisType\_Class\_Name\_\_define.pro* must exist somewhere in IDL's path for the visualization type to be successfully registered.

---

For example, the following method call registers a visualization type named `myVis` for which the class definition is stored in the file `myVisualization__define.pro`:

```
self -> RegisterVisualization, 'myVis', 'myVisualization'
```

See [“Registering a Visualization Type”](#) on page 110 for additional details. See [“Predefined iTool Visualization Classes”](#) on page 91 for a list of visualization types included in the iTool system as installed with IDL.

## Registering Operations

Registering an operation with an iTool class allows instances of the iTool to apply the registered operation to data selected in the iTool. Any number of operations can be registered with a given iTool.

Operations are registered by calling the `IDLitool::RegisterOperation` method:

```
self -> RegisterOperation, Operation_Type, OpType_Class_Name, $
      IDENTIFIER = identifier
```

where *Operation\_Type* is the string you will use when referring to the operation, *OpType\_Class\_Name* is a string that specifies the name of the class file that contains the operation's definition, and *identifier* is a string containing the operation's iTool identifier. (The identifier is used to specify where on the iTool's menu bar the operation will appear. See [“iTool Object Identifiers”](#) on page 19 for a discussion of iTool system identifiers.)

### Note

---

The file *OpType\_Class\_Name\_\_define.pro* must exist somewhere in IDL's path for the visualization type to be successfully registered.

---

For example, the following method call registers an operation named `myOp` for which the class definition is stored in the file `myOperation__define.pro`, and places the menu selection `Change My Data` in the `Filters` folder of the iTool Operations menu.

```
self -> RegisterVisualization, 'myOp', 'myOperation', $
      IDENTIFIER = 'Operations/Filters/Change My Data'
```

See [“Registering an Operation”](#) on page 153 for additional details. See [“Predefined iTool Operations”](#) on page 122 for a list of operations included in the iTool system as installed with IDL.

## Registering File Readers and Writers

Registering a file reader or file writer with an iTool class allows instances of the iTool to read or write files of the type handled by the reader or writer. Any number of file readers and writers can be registered with a given iTool.

File readers are registered by calling the `IDLitool::RegisterFileReader` method:

```
self -> RegisterFileReader, Reader_Type, ReaderType_Class_Name, $
      ICON = icon
```

where *Reader\_Type* is the string you will use when referring to the file reader, *ReaderType\_Class\_Name* is a string that specifies the name of the class file that



contains the file writer's definition, and *icon* is a string containing the name of a bitmap file used to represent the file reader.

Similarly, file writers are registered by calling the `IDLitool::RegisterFileWriter` method:

```
self -> RegisterFileWriter, Writer_Type, WriterType_Class_Name, $
      ICON = icon
```

where *Reader\_Type* is the string you will use when referring to the file reader, *ReaderType\_Class\_Name* is a string that specifies the name of the class file that contains the file writer's definition, and *icon* is a string containing the name of a bitmap file used to represent the file writer. See [“Icon Bitmaps”](#) on page 28 for details on where bitmap icon files are located.

---

### Note

The class definition files *ReaderType\_Class\_Name\_\_define.pro* or *WriterType\_Class\_Name\_\_define.pro* must exist somewhere in IDL's path for the file reader or writer to be successfully registered.

---

For example, the following method call registers a file reader named `myReader` for which the class definition is stored in the file `myFileReader__define.pro`, and specifies the file `reader.bmp` located in the `home/mydir` directory as the icon to use on the toolbar.

```
self -> RegisterFileReader, 'myReader', 'myFileReader', $
      ICON = '/home/mydir/reader.bmp'
```

See [“Registering a File Reader”](#) on page 177 for additional details. See [“Predefined iTool File Readers”](#) on page 163 for a list of file readers included in the iTool system as installed with IDL.

Similarly, the following method call registers a file writer named `myWriter` for which the class definition is stored in the file `myFileWriter__define.pro`, and specifies the file `writer.bmp` located in the `home/mydir` directory as the icon to use on the toolbar.

```
self -> RegisterFileReader, 'myWriter', 'myFileWriter', $
      ICON = '/home/mydir/writer.bmp'
```

See [“Registering a File Writer”](#) on page 201 for additional details. See [“Predefined iTool File Writers”](#) on page 187 for a list of file writers included in the iTool system as installed with IDL.

## Example Init Method

The following example code shows a very simple Init method for an iTool named `ExampleTool`. This function should be included in a file named `ExampleTool__define.pro`.

```

FUNCTION ExampleTool::Init, _REF_EXTRA = _EXTRA

; Call the Init method of the super class.
IF (self -> IDLitToolbase::Init(NAME='ExampleTool', $
    DESCRIPTION = 'Example Tool Class', _EXTRA = _extra) EQ 0) THEN $
    RETURN, 0

; Register a visualization
self -> RegisterVisualization, 'Image', 'IDLitVisImage', $
    ICON = 'image'

; Register an operation
self -> RegisterOperation, 'Byte Scale', 'IDLitOpBytScl', $
    IDENTIFIER = 'Operations/Byte Scale'

RETURN, 1

END

```

## Discussion

The `ExampleTool` is based on the `IDLitToolbase` class (discussed in [“Subclassing from the IDLitToolbase Class”](#) on page 75). As a result, all of the standard iTool operations, manipulators, file readers and writers are already present. The `ExampleTool` Init method needs to do only three things:

1. Call the Init method of the superclass, `IDLitToolbase`, using the `_EXTRA` keyword inheritance mechanism to pass through any keywords provided when the `ExampleTool` Init method is called.
2. Register a visualization type for the tool. We choose the standard image visualization defined by the `idlitvisimage__define.pro` class definition file,
3. Register an operation. We choose an operation that implements the IDL `BYTSCL` function, defined by the `idlitopbytsc1__define.pro` class definition file and place a menu item in the iTool Operations menu.

**Note**

This example is intended to demonstrate how simple it can be to create a new iTool class definition. While the class definition for an iTool with significant extra functionality will register more features, the process is the same.

## Unregistering Components

In some cases, you may want to subclass from an iTool class that contains features you do not want to include in your class. Rather than building a class that duplicates most, but not all, of the functionality of the existing class, you can create a subclass that explicitly *unregisters* the components that you don't want included.

For each Register method of the IDLitTool class there is a corresponding UnRegister method. Call the UnRegister method with the *Name* you used when registering the component. For example, if your superclass registers an operation with the identifier 'MultiplyBy100' and you do not want this operation included in your class, you would include the following method call in your iTool class Init method:

```
self -> UnRegisterOperation, 'MultiplyBy100'
```

## Creating the Class Structure Definition

When any IDL object is created, IDL looks for an IDL class structure definition that specifies the instance data fields needed by an instance of the object, along with the data types of those fields. The object class structure must have been defined *before* any objects of the type are created. In practice, when the IDL OBJ\_NEW function attempts to create an instance of a specified object class, it executes a procedure named *ObjectClass\_\_define* (where *ObjectClass* is the name of the object), which is expected to define an IDL structure variable with the correct name and structure fields. For additional information on how IDL creates object instances, see “[The Object Lifecycle](#)” in Chapter 22 of the *Building IDL Applications* manual.

## Subclassing from the IDLitToolbase Class

The IDLitToolbase class defines the base operations and user interface functionality used in iTools created by RSI. If your aim is to create an iTool that has base functionality similar to that included in the standard iTools, you will want to subclass from the IDLitToolbase class, or from another tool that subclasses from the IDLitToolbase class.

The IDLitToolbase class registers a large number of operations, manipulators, file readers, and file writers. This base feature set may change from release to release;

inspect the file `idlitoolbase__define.pro` in the `lib/itools` subdirectory of your IDL distribution for the exact set of features included in your distribution.

---

### Note

To create an iTool that *does not* include the standard iTool functionality, subclass from the `IDLitTool` class.

---

In general, the `IDLitToolbase` class registers the following types of features:

**Standard menu items** — Operations that appear in the **File**, **Edit**, **Insert**, **Window**, and **Help** menus are defined in the `IDLitToolbase` class. If you are building a subclass of the `IDLitToolbase` class, you have the option of adding items to or removing items from these menus in your own class definition file.

**Operations menu items** — Standard data-centric operations provided as part of the iTools distribution and which appear in all of the standard iTools are placed on the **Operations** menu by the `IDLitToolbase` class.

**Context menu items** — Standard operations such as Cut, Copy, Paste, Group, Ungroup, *etc.* are included on the context menu by the `IDLitToolbase` class.

**Toolbar items** — Operations that enable standard File and Edit menu functionality are placed on the toolbar by the `IDLitToolbase` class. In addition, standard manipulators (zoom, arrow, and rotate), and annotations (text, line, rectangle, oval, polygon, and freeform) are placed on the toolbar.

**File readers** — All file readers included in the iTools distribution are registered by the `IDLitToolbase` class. File readers do not appear in the iTool interface, but are used automatically when importing a data file.

**File writers** — All file writers included in the iTools distribution are registered by the `IDLitToolbase` class. File writers do not appear in the iTool interface, but are used automatically when exporting data to a file.

## Example Class Structure Definition

The following is a very simple iTool class structure definition for an iTool named `ExampleTool`. This procedure should be the last procedure in a file named `exampletool__define.pro`.

```
PRO ExampleTool__Define
  struct = { ExampleTool,          $
            INHERITS IDLitToolbase $ ; Provides iTool interface
          }
END
```

## Discussion

The purpose of the structure definition routine is to define a named IDL structure with structure fields that will contain the iTool object instance data. The structure name should be the same as the iTool's class name — in this case, `ExampleTool`.

Like many iTools, `ExampleTool` is created as a subclass of the `IDLitToolbase` class. iTools that subclass from `IDLitToolbase` inherit all of the standard iTool functionality, as described in [“Subclassing from the IDLitToolbase Class”](#) on page 75.

---

## Note

This example is intended to demonstrate how simple it can be to create a new iTool class definition. While the class definition for an iTool with significant extra functionality will likely define additional structure fields, and may inherit from other iTool classes, the basic principles are the same.

---

# Registering a New Tool Class

Before an instance of a new iTool can be created, the tool's class definition must be registered with the iTool system. Registering an iTool class with the system links the class definition file containing the actual IDL code that initializes an iTool object with a simple string that names the iTool. Since you use the name string in code that creates instances of individual tools, a change to the name of the class definition file requires only a change to the code that registers the iTool class.

iTool classes are registered using the ITREGISTER procedure. In most cases, the call to the ITREGISTER procedure will be included in an iTool's launch routine, but the call can take place in any code at any time. If multiple iTool launch routines create instances of the same iTool class, however, you may find it more convenient to register the iTool in a single routine, called only once. This removes the need to call the registration routine in each launch routine individually.

---

**Note**

If only a small number of routines will create instances of a given iTool, you may find it more convenient to register the iTool class within the tool launch routine.

---

## Using ITREGISTER

Use the ITREGISTER routine to register the class definition:

```
ITREGISTER, 'Tool Name', 'Tool_Class_Name'
```

where *Tool Name* is a string you will use to create instances of the tool, and *Tool\_Class\_Name* is a string that specifies the name of the class file that contains the tool's definition.

---

**Note**

The file *Tool\_Class\_Name\_\_define.pro* must exist somewhere in IDL's path for the tool definition to be successfully registered.

---

If a given iTool class has already been registered when the ITREGISTER routine is called, the class will not be registered a second time. The registration can be performed at any time in an IDL session before you attempt to create an instance of the iTool.

See **"ITREGISTER"** in the *IDL Reference Guide* manual for details.

## Example

Suppose you have an iTool class definition file named `myTool__define.pro`, located in a directory included in IDL's `!PATH` system variable. Register this class with the iTool system with the following command:

```
ITREGISTER, 'My First Tool', 'myTool'
```

Tools defined by the `myTool` class definition file can now be created by the iTool system by specifying the tool name `My First Tool`. In most cases, this command would be included in the launch routine for the `myTool` iTool, but the call can be placed in any code that is executed before the first instance of the iTool is created.

# Creating an iTool Launch Routine

An iTool launch routine is an IDL procedure that creates an instance of an iTool by calling the `IDLITSYS_CREATETOOL` function. The launch routine may do other things as well, including creating data objects to pass to the create function from command-line arguments.

The process of creating an iTool launch routine is outlined in the following sections:

- “[Specifying Command-Line Arguments and Keywords](#)” on page 80
- “[Creating Data Objects](#)” on page 81
- “[Handling Errors](#)” on page 82
- “[Creating an iTool Instance](#)” on page 83

## Specifying Command-Line Arguments and Keywords

If you want to be able to specify data to be loaded into your iTool when launching the tool from the IDL command line, you must specify positional arguments or keywords in the procedure definition. The procedure definition for an iTool launch routine may look something like the following:

```
PRO myTool, A1, A2, MYKEYWORD = myKeys, IDENTIFIER = id, $
    _EXTRA = _extra
```

Here, there are two positional parameters (or arguments) and three keyword parameters are specified.

### Arguments

Data arguments are specified in an iTool launch routine as with any IDL procedure. See “[Parameters](#)” in Chapter 4 of the *Building IDL Applications* manual for details on arguments.

### Keywords

Keyword arguments to an iTool launch routine are handled as with any IDL procedure. See “[Parameters](#)” in Chapter 4 of the *Building IDL Applications* manual for details on keyword arguments. In addition, you may want to include the following keyword arguments in the definition of the launch routine:

#### The IDENTIFIER Keyword

The `IDENTIFIER` keyword is used to return the iTool system identifier string for the newly created tool. You must set the variable specified by the `IDENTIFIER` keyword



equal to the return value of the `IDLITSYS_CREATETOOL` function. This allows the user to retrieve the newly-created iTool's identifier in an IDL variable by including the `IDENTIFIER` keyword in the call to the launch routine. The iTool identifier can then be used to specify the iTool as the target for another operation, such as overplotting.

### The `_EXTRA` Keyword

Optionally, you can use IDL's keyword inheritance mechanism to pass keyword parameters that are not explicitly handled by your routine through to other routines. See "[Keyword Inheritance](#)" in Chapter 4 of the *Building IDL Applications* manual for details on IDL's keyword inheritance mechanism.

## Creating Data Objects

If your iTool launch routine allows users to specify data arguments (as opposed to keywords that are passed through to the iTool component objects), you must process those arguments and create an `IDLitParameterSet` object to be passed to the `IDLITSYS_CREATETOOL` function. Parameter sets, data types, and general iTool system data handling concepts are discussed in detail in [Chapter 3, "Data Management"](#).

### Parameter Sets

Data is passed to a newly-created iTool instance by supplying an `IDLitParameterSet` object as the value of the `INITIAL_DATA` keyword to the `IDLITSYS_CREATETOOL` function. To create a parameter set object, use the `OBJ_NEW` function:

```
oParameterSet = OBJ_NEW('IDLitParameterSet', NAME = paramSetName)
```

where `oParameterSet` is a named variable that will hold the object reference to the parameter set object and `paramSetName` is a string that will be used by the iTool user interface to refer to the parameter set.

For example, if you are creating a data container to hold X and Y vectors to be plotted in two-dimensions, you might use the following code:

```
oPlotData = OBJ_NEW('IDLitParameterSet', NAME = 'Plot data')
```

See [Chapter 3, "Data Management"](#), and "[IDLitParameterSet](#)" in the *IDL Reference Guide* manual for details.

### Data Items

The parameter set object holds objects of type `IDLitData`, or objects of types derived from `IDLitData`, such as `IDLitDataImage` or `IDLitDataVector`. These data objects, in

turn, hold the actual data used by the iTool. To create a data object, use the `OBJ_NEW` function:

```
oData = OBJ_NEW('IDLitData', vData, TYPE = dataType, $
    NAME = dataName)
```

where `oData` is a named variable that will hold the object reference to the data object, `vData` is an IDL variable containing the actual data, `dataType` is a string specifying the iTool data type of the data held by the object, and `dataName` is a string that will be used by the iTool user interface to refer to the data object. See [“iTool Data Types”](#) on page 34 for additional information on iTool data types.

For example, if you are creating a data object to hold the Y vector of a two-dimensional plot, you might use the following code:

```
oPlotY = OBJ_NEW('IDLitData', yData, TYPE = 'IDLVECTOR', $
    NAME = 'Y data')
```

Here, the data that make up the Y vector are contained in the variable `yData`. After a data item has been created, it must be added to the parameter set object. Continuing our example, the following code adds the `oPlotY` data object to the `oPlotData` parameter set object, assigning the parameter name 'Y data':

```
oPlotData -> Add, oPlotY, PARAMETER_NAME='Y data'
```

See [Chapter 3, “Data Management”](#), and [“IDLitData”](#) in the *IDL Reference Guide* manual for details.

## Example

For an example iTool launch routine that creates and populates a parameter set object, see [“Example: Simple iTool”](#) on page 85.

## Handling Errors

The error-handling requirements of your iTool launch routine will depend largely on the type of data processing you perform. In general, your goal should be to clean up any objects or pointers your routine creates, display an error message to the user, and return to the calling routine. It is beyond the scope of this chapter to discuss IDL’s error handling mechanisms in detail; for more information see [Chapter 18, “Controlling Errors”](#) in the *Building IDL Applications* manual.

iTool launch routines included in the IDL distribution handle errors by placing a block of IDL code that looks like the following at the beginning of the routine:

```
ON_ERROR, 2
CATCH, iErr
IF (iErr NE 0) THEN BEGIN
```

```

    CATCH, /CANCEL
    IF OBJ_VALID(oDataObject) THEN OBJ_DESTROY, oDataObject
    MESSAGE, /REISSUE_LAST
    RETURN
ENDIF

```

This block of error-handling code does the following:

1. Uses the ON\_ERROR procedure to instruct IDL to return to the caller of the program that establishes an error condition.
2. Uses the CATCH procedure to establish an error-handler for the iTool launch routine, returning the error code in the variable *iErr*.
3. If the value of *iErr* is not 0 (that is, if an error is detected), do the following:
  - Use the CATCH procedure again to cancel the error handler.
  - Destroy any data objects created by the launch routine. In most cases, destroying the data container object (represented here by *oDataObject*) will be sufficient to destroy all objects added to the data container.
  - Use the MESSAGE routine to display the error message in the IDL output log.

Once these tasks have been accomplished, use the RETURN procedure to return to the routine that called the iTool launch routine, or to the IDL Main level, if the launch routine was invoked at the IDL command prompt.

Depending on the complexity of your iTool launch routine, additional cleanup may be required. For example, you may need to free IDL pointers created by the launch routine. In many cases, however, error-handling code similar to that used in the standard iTool launch routines will be sufficiently robust.

## Creating an iTool Instance

Create an instance of your iTool class by calling the IDLITSYS\_CREATETOOL function:

```

id = IDLITSYS_CREATETOOL('Tool Name', NAME = 'Tool Label', $
    VISUALIZATION_TYPE = 'VisType', $
    INITIAL_DATA = 'oDataContainer', _EXTRA = _extra)

```

where *Tool Name* is the name of a previously-registered iTool class, *Tool Label* is a text label that will be used in the iTool user interface to identify this instance of the iTool, *VisType* is the name of a previously-registered iTool visualization type (or array of visualization types), and *oDataContainer* is an IDLItDataContainer object created from the values specified as arguments or keywords.

We also use IDL's keyword inheritance mechanism (the `_EXTRA` keyword) to pass any additional keyword parameters specified when the launch routine is called through to the lower-level iTool routines.

See “[IDLITSYS\\_CREATETOOL](#)” in the *IDL Reference Guide* manual for details.

### **iTool Class Registration**

Before an instance of an iTool can be created, the iTool class must be registered with the iTool system. An iTool class can be registered with the system within the launch routine by calling the `ITREGISTER` routine, but you may benefit from registering iTool classes separately. See “[Registering a New Tool Class](#)” on page 78 for details.

### **iTool Visualization Type Registration**

Similarly, the visualization type or types specified by the `VISUALIZATION_TYPE` keyword must have been registered with the system. In most cases, visualizations will either be predefined iTool visualizations (see “[Predefined iTool Visualization Classes](#)” on page 91) or will be registered in the iTool class' `Init` method, as described in “[Creating a New iTool Class](#)” on page 69. All iTools must have at least one visualization type. Multiple visualization types are specified by supplying a string array as the value of the `VISUALIZATION_TYPE` property.

#### **Note**

---

Once a visualization type has been registered with the iTool system, it is available to *all* iTools launched during an IDL session. This means that the list of visualization types available to a given iTool can change if other iTools are launched.

---

# Example: Simple iTool

This example creates a very simple iTool named `example1tool` that incorporates standard functionality from the iTools distribution.

## Class Definition File

The class definition for the `example1tool` consists of an `Init` method and a class structure definition routine. As with all object class definition files, the class structure definition routine is the last routine in the file, and the file is given the same name as the class definition routine (with the suffix `.pro` appended).

### Init Method

```
FUNCTION example1tool::Init, _REF_EXTRA = _EXTRA

; Call our super class
IF ( self -> IDLitToolbase::Init(_EXTRA = _extra) EQ 0) THEN $
    RETURN, 0

;*** Visualizations
self -> RegisterVisualization, 'Image', 'IDLitVisImage', $
    ICON = 'image', /DEFAULT

self -> RegisterVisualization, 'Colorbar', 'IDLitVisColorbar', $
    ICON = 'colorbar'

;*** Insert menu
self -> RegisterOperation, 'Colorbar', 'IDLitOpInsertColorbar', $
    IDENTIFIER = 'Insert/Colorbar', ICON = 'colorbar'

RETURN, 1

END
```

### Discussion

The first item in our class definition file is the `Init` method. The `Init` method's function signature is defined first, using the class name `example1tool`. Note the use of the `_REF_EXTRA` keyword inheritance mechanism; this allows any keywords specified in a call to the `Init` method to be passed through to routines that are called within the `Init` method even if we do not know the names of those keywords in advance.

Next, we call the `Init` method of the superclass. In this case, we are creating a subclass of the `IDLitToolbase` class; this provides us with all of the standard iTool functionality automatically. Any “extra” keywords specified in the call to our `Init`

method are passed to the `IDLitToolbase::Init` method via the keyword inheritance mechanism.

We register two standard iTool visualization types: `Image` and `Colorbar`. Both of these types are part of the regular iTool distribution, so we simply register the existing classes.

We also register a standard iTool operation: `Insert Colorbar`. Our call to the `RegisterOperation` method specifies the `IDENTIFIER` property as `'Insert/Colorbar'`, which places a **Colorbar** entry on the **Insert** menu of the iTool.

Finally, we return the value 1 to indicate successful initialization.

## Class Definition

```
PRO exampleltool__Define

struct = { exampleltool,          $
           INHERITS IDLitToolbase $ ; Provides iTool interface
        }

END
```

## Discussion

Our class definition routine is very simple. We create an IDL structure variable with the name `exampleltool`, specifying that the structure inherits from the `IDLitToolbase` class.

## Launch Routine

Our iTool launch routine also uses the class name `exampleltool`. It accepts a single data argument, which we assume will contain an image array. The code is shown below:

```
PRO exampleltool, data, IDENTIFIER = IDENTIFIER, _EXTRA = _EXTRA

nParams = N_PARAMS()

IF (nParams gt 0) THEN BEGIN
  oParmSet = OBJ_NEW('IDLitParameterSet', $
    NAME = 'example 1 parameters', $
    ICON = 'image', $
    DESCRIPTION = 'Example tool parameters')
```

```

IF (N_ELEMENTS(data) GT 0) THEN BEGIN
  oData = OBJ_NEW("IDLitDataIDLImagePixels")
  result = oData -> SetData(data, _EXTRA = _EXTRA)
  oParmSet -> Add, oData, PARAMETER_NAME = "ImagePixels"

  ; Create a default grayscale ramp.
  ramp = BINDGEN(256)
  oPalette = OBJ_NEW('IDLitDataIDLPalette', $
    TRANPOSE([[ramp], [ramp], [ramp]]), $
    NAME = 'Palette')
  oParmSet -> Add, oPalette, PARAMETER_NAME = 'PALETTE'

ENDIF

ENDIF

ITREGISTER, "Example 1 Tool", "example1tool"

identifier = IDLITSYS_CREATETOOL("Example 1 Tool", $
  VISUALIZATION_TYPE = ["Image"], $
  INITIAL_DATA = oParmSet, _EXTRA = _EXTRA, $
  TITLE = "First Example iTool")

END

```

## Discussion

Our iTool launch routine accepts a single *data* argument. We also specify that our launch routine should accept the IDENTIFIER keyword; we will use the variable specified as the value of this keyword (if any) to return the iTool identifier of the new iTool we create.

First, we check the number of non-keyword arguments that were supplied using the N\_PARAMS function. If an argument was supplied, we create an IDLitParameterSet object to hold the data.

Next, we check to make sure the supplied data argument is not empty using the N\_ELEMENTS function. If the supplied argument contains data, we create an IDLitDataImage object to contain the image data and add the object to our parameter set object, assigning the parameter name 'Image'.

## Note

---

In the interest of brevity, we do very little data verification in this example. We could, for example, verify that the data argument contains a two-dimensional array of a specified type.

---

We use the ITREGISTER procedure to register our iTool class with the name "Example 1 Tool".

Finally, we call the `IDLITSYS_CREATETOOL` function with the registered name of our `iTool` class.





# Chapter 6: Creating a Visualization

This chapter describes the process of creating an iTool visualization type.

---

<a href="#">Overview</a> .....	90	<a href="#">Registering a Visualization Type</a> .....	110
<a href="#">Predefined iTool Visualization Classes</a> ....	91	<a href="#">Unregistering a Visualization Type</a> .....	112
<a href="#">Creating a New Visualization Type</a> .....	95	<a href="#">Example: Image-Contour Visualization</a> ..	113

# Overview

A *visualization type* is an iTool component object class that contains core IDL graphic objects (IDLgrPlot objects, for example), other iTool visualization components, or both. Visualization type components can also contain data. A number of visualization types are predefined and included in the IDL iTools package; if none of the predefined types suits your needs, you can create your own visualization type by subclassing either from one of the predefined types or from the base IDLitVisualization class on which all of the predefined types are based.

## The Visualization Type Creation Process

To create a new iTool visualization type, you will do the following:

- Choose an iTool visualization class on which your new visualization type will be based. In almost all cases, you will base new visualization types either on the IDLitVisualization class or on a visualization class that is itself based on IDLitVisualization. The IDLitVisualization class automatically handles selection, selection visuals, data ranges, and notification of data changes.
- Define the data parameters necessary to create a visualization of the new type.
- Define the properties of the visualization, and set default property values.
- Override methods used to get or set properties, react to changes in the underlying data, and clean up, as necessary.

This chapter describes the process of creating a new visualization type based on the IDLitVisualization class.

# Predefined iTool Visualization Classes

The iTool system distributed with IDL includes a number of pre-defined visualization classes. You can include these visualization classes in an iTool directly by registering the class with your iTool (as described in “[Registering a Visualization Type](#)” on page 110). You can also create a new visualization class based on one of the pre-defined classes. Visualization classes are located in the `lib/itools/components` subdirectory of the IDL directory.

## IDLitVisAxis

Displays a single axis object.

### Data Types Accepted

- None

## IDLitVisColorbar

Displays a color bar.

### Data Types Accepted

- Palette data: IDLPALETTE

## IDLitVisContour

Displays a two-dimensional or three-dimensional contour plot.

### Data Types Accepted

- Z data: IDLARRAY2D
- X and Y data: IDLVECTOR

## IDLitVisHistogram

Displays a histogram plot of the input data.

### Data Types Accepted

- Histogram data: IDLVECTOR, IDLARRAY2D, IDLARRAY3D

## IDLitVisImage

Displays an image.

**Data Types Accepted**

- Image data: IDLIMAGE, IDLARRAY2D
- Palette data: IDLPALETTE, IDLARRAY2D

**IDLitVisIsosurface**

Displays an isosurface created from existing volume data.

**Data Types Accepted**

- None

**IDLitVisLegend**

Displays a legend that can contain multiple IDLitVisLegendContourItem, IDLitVisLegendPlotItem, and IDLitVisLegendSurfaceItem objects.

**Data Types Accepted**

- None

**IDLitVisLight**

Places a light object in the iTool visualization window to illuminate surface and volume objects.

**Data Types Accepted**

- None

**IDLitVisPlot**

Displays a two-dimensional line plot.

**Data Types Accepted**

- X and Y data: IDLVECTOR
- Vertex data: IDLARRAY2D
- X and Y error data: IDLVECTOR, IDLARRAY2D

**IDLitVisPlot3D**

Displays a two-dimensional line plot.

**Data Types Accepted**

- X, Y, and Z data: IDLVECTOR

- Vertex data: IDLARRAY2D
- X, Y, and Z error data: IDLVECTOR, IDLARRAY2D

## **IDLitVisPolygon**

Displays a polygon annotation

### **Data Types Accepted**

- Vertex data: IDLARRAY2D

## **IDLitVisPolyline**

Displays a single polyline.

### **Data Types Accepted**

- Vertex data: IDLARRAY2D

## **IDLitVisRoi**

Defines and displays a polygonal region of interest.

### **Data Types Accepted**

- Vertex data: IDLARRAY2D

## **IDLitVisSurface**

Displays a three-dimensional surface plot.

### **Data Types Accepted**

- Z (surface) data: IDLARRAY2D
- X and Y data: IDLVECTOR, IDLARRAY2D
- Vertex color data: IDLVECTOR, IDLARRAY2D
- Texture maps: IDLARRAY3D, IDLARRAY2D
- Palette colors: IDLARRAY2D

## **IDLitVisText**

Displays text string.

### **Data Types Accepted**

- Location data: IDLVECTOR

## **IDLitVisVolume**

Displays a three-dimensional volume rendering.

### **Data Types Accepted**

- Volume data: IDLARRAY3D
- Palette data: IDLPALETTE
- Opacity table data: IDLOPACITY\_TABLE

# Creating a New Visualization Type

An iTool visualization class definition file must (at the least) provide methods to initialize the visualization class, get and set property values, handle changes to the underlying data, clean up when the visualization is destroyed, and define the visualization class structure. Complex visualization types will likely provide additional methods.

The process of creating a visualization type is outlined in the following sections:

- [“Creating an Init Method”](#) on page 95
- [“Creating a Cleanup Method”](#) on page 102
- [“Creating a GetProperty Method”](#) on page 103
- [“Creating a SetProperty Method”](#) on page 104
- [“Creating an onDataChangeUpdate Method”](#) on page 105
- [“Creating an onDataDisconnect Method”](#) on page 107
- [“Creating the Class Structure Definition”](#) on page 107

## Creating an Init Method

The visualization class Init method handles any initialization required by the visualization object, and should do the following:

- define the Init function method
- call the Init methods of any superclasses
- register any data parameters used when creating visualizations of the new type
- register any properties of your visualization type, and set property attributes as necessary
- create all the graphics objects needed by the visualization, and add them to the visualization object
- define a custom selection visual, if desired
- perform other initialization steps as necessary
- return the value 1 if the initialization steps are successful, or 0 otherwise

**Note**


---

While the `Init` method *registers* data parameters for a visualization, it does not *accept* data parameters itself. Data parameters are set in the `OnDataChangeUpdate` method.

---

**Definition of the Init Function**

Begin by defining the argument and keyword list for your `Init` method. The argument and keyword list defines positional parameters (arguments) accepted by your method, defines any keywords that will be handled directly by your method, and specifies whether keywords not explicitly handled by your method will be passed through to other routines called by your method via IDL's keyword inheritance mechanism. The `Init` method for a visualization type generally looks something like this:

```
FUNCTION MyVisualization::Init, MYKEYWORD1 = mykeyword1, $
    MYKEYWORD2 = mykeyword2, ..., _REF_EXTRA = _extra
```

where *MyVisualization* is the name of your visualization class and the *MYKEYWORD* parameters are keywords handled explicitly by your `Init` function.

Use keyword inheritance (the `_REF_EXTRA` keyword) to pass keyword parameters through to any called routines as necessary. See “[Keyword Inheritance](#)” in Chapter 4 of the *Building IDL Applications* manual for details on IDL's keyword inheritance mechanism.

**Superclass Initialization**

The visualization class `Init` method should call the `Init` method of any required superclass. For example, if your visualization class is based on an existing visualization, you would call that visualization's `Init` method:

```
success = self -> SomeVisualizationClass::Init(_EXTRA = _extra)
```

where *SomeVisualizationClass* is the class definition file for the visualization on which your new visualization is based. The variable `success` will contain a 1 if the initialization is successful.

**Note**


---

Your visualization class may have multiple superclasses. In general, each superclass' `Init` method should be invoked by your class' `Init` method.

---

**Error Checking**

Rather than simply calling the superclass `Init` method, it is a good idea to check whether the call to the superclass `Init` method succeeded. The following statement



checks the value returned by the superclass Init method; if the returned value is 0 (indicating failure), the current Init method also immediately returns with a value of 0:

```
IF (self -> SomeVisualizationClass::Init() EQ 0) THEN RETURN, 0
```

This convention is used in all visualization classes included with IDL. RSI strongly suggests that you include similar checks in your own class definition files.

## Keywords to the Init Method

Properties of the visualization type class can be set in the Init method by specifying the property names and values as IDL keyword-value pairs. In addition to any keywords implemented directly in the Init method of the superclass on which you base your class, the properties of the IDLitVisualization class are available to any visualization class. See “[IDLitVisualization Properties](#)” in the *IDL Reference Guide* manual.

Use keyword inheritance (the `_EXTRA` keyword) to pass keyword parameters through to the superclass as necessary. See “[Keyword Inheritance](#)” in Chapter 4 of the *Building IDL Applications* manual for details on IDL’s keyword inheritance mechanism.

## Standard Base Class

While you can create your new visualization class from any existing visualization class, in many cases, visualization classes you create will be subclassed directly from the base class IDLitVisualization:

```
IF (self -> IDLitVisualization::Init(_EXTRA = _extra) EQ 0) $  
THEN RETURN, 0
```

The IDLitVisualization class provides the base iTool functionality used in the visualization classes created by RSI. See “[Subclassing from the IDLitVisualization Class](#)” on page 108 for details.

## Return Value

If all of the routines and methods used in the Init method execute successfully, the method should indicate successful initialization by returning 1. Other visualization classes that subclass from your visualization class may check this return value, as your routine should check the value returned by any superclass Init methods called.

## Registering Parameters

Visualization types must register each data parameter used to create the visualization. Data parameters are described in detail in [Chapter 3, “Data Management”](#).

Register a parameter by calling the `RegisterParameter` method of the `IDLitParameter` class:

```
self -> RegisterParameter, ParmameterName, $
    TYPES = ['DataType1', ..., 'DataTypeN']
```

where *ParameterName* is a string that defines the name of the parameter and the `TYPES` keyword is set equal to a string or array of strings specifying the iTool system data types the parameter can represent. See “[Registering Parameters](#)” on page 41 for additional details.

## Registering Properties

Visualization types can register properties with the iTool; registered properties show up in the property sheet interface, and can be modified interactively by users. The iTool property interface is described in detail in [Chapter 4, “Property Management”](#).

Register a property by calling the `RegisterProperty` method of the `IDLitComponent` class:

```
self -> RegisterProperty, PropertyIdentifier [, TypeCode] $
    [, ATTRIBUTE = value]
```

where *PropertyIdentifier* is a string that uniquely identifies the property, *TypeCode* is an integer between 0 and 9 specifying the property data type, and *ATTRIBUTE* is a property attribute. See “[Registering Properties](#)” on page 54 for details.

### Property Aggregation

IDL objects can *contain* other objects; a visualization type is, at one level, simply an object container that holds the different graphics objects that make up a visualization. The iTools *property aggregation* mechanism allows the properties of several different objects held by the same container object to be displayed in the same property sheet automatically. Without property aggregation, you would have to manually register all of the properties of the objects contained in your visualization type object.

Aggregate the properties of contained objects using the `Aggregate` method of the `IDLitVisualization` class:

```
self -> Aggregate, Object_Reference
```

where *Object\_Reference* is a reference to the object whose properties you want aggregated into the visualization object. See “[Property Aggregation](#)” on page 61 for additional details.

### Note

The `IDLitVisualization::Add` method includes an `AGGREGATE` keyword. This keyword is simply a shorthand method of aggregating the properties of an object

during the call to the Add method, eliminating the need to call the Aggregate method separately. The call

```
self -> Add, Object_Reference, /AGGREGATE
```

is the same as the following two calls:

```
self -> Add, Object_Reference  
self -> Aggregate, Object_Reference
```

---

## Setting Property Attributes

If a property has already been registered, perhaps by a superclass of your visualization class, you can change the registered attribute values using the SetPropertyAttribute method of the IDLitComponent class:

```
self -> SetPropertyAttribute, Identifier
```

where *Identifier* is the name of the keyword to the GetProperty and SetProperty methods used to retrieve or change the value of this property. (The Identifier is specified in the call to RegisterProperty either via the *PropertyName* argument or the IDENTIFIER keyword.) See “[Property Attributes](#)” on page 58 for additional details.

## Adding Graphics Objects to the Visualization

An iTool visualization type must contain at least one IDLit\* visualization object or IDLgr\* graphics object. To add a visualization or graphics object, you must first create an instance of the object using the OBJ\_NEW function, then add the object instance to the visualization using the Add method of the IDLitVisualization class:

```
Graphics_Object = OBJ_NEW('IDLitVisObject')  
self -> Add, Graphics_Object
```

where *IDLitVisObject* is an actual IDL iTool visualization class, such as IDLitVisPlot.

In practice, you should also consider the following when adding a visualization or graphics object to a visualization type:

- The visualization or graphics object reference should generally be placed in a specific field of the visualization type’s class structure. This allows you access to the object when you have the reference to the visualization object itself.
- In most cases, you will want to include the REGISTER\_PROPERTIES keyword in the call to OBJ\_NEW when creating a visualization or graphics object instance. This keyword does the work of registering all registrable

properties of the object automatically, relieving you from the need to manually register the properties you want to show up in the visualization's property sheet.

A typical addition of a graphics object to a visualization looks like this:

```
self._oPlot = OBJ_NEW('IDLitVisPlot', /REGISTER_PROPERTIES)
self -> Add, self._oPlot, /AGGREGATE
```

Here, we create a new `IDLitVisPlot` object instance and place the object reference in the `_oPlot` field of the visualization's class structure. The `REGISTER_PROPERTIES` keyword ensures that all of the registrable `IDLitVisPlot` properties are registered with the visualization automatically. Next, we use the `Add` method to add the object instance to our visualization; this inserts the object into the visualization's graphics hierarchy. Finally, we use the `AGGREGATE` keyword to include all of the `IDLitVisPlot` object's registered properties in the visualization's property sheet.

## Passing Through Caller-Supplied Property Settings

If you have included the `_REF_EXTRA` keyword in your function definition, you can use IDL's keyword inheritance mechanism to pass any "extra" keyword values included in the call to the `Init` method through to other routines. One of the things this allows you to do is specify property settings when the `Init` method is called; simply include each property's keyword/value pair when calling the `Init` method, and include the following in the body of the `Init` method:

```
IF (N_ELEMENTS(_extra) GT 0) THEN $
  self -> MyVisualization::SetProperty, _EXTRA = _extra
```

where *MyVisualization* is the name of your visualization class. This line has the effect of passing any "extra" keyword values to your visualization class' `SetProperty` method, where the keyword can either be handled directly or passed through to the `SetProperty` methods of the superclasses of your class. See ["Creating a SetProperty Method"](#) on page 104 for details.

## Example Init Method

The following example code shows a very simple `Init` method for a visualization type named `ExampleVis`. This function would be included (along with the class structure definition routine and any other methods defined by the class) in a file named `examplevis__define.pro`.

```
FUNCTION ExampleVis::Init, _REF_EXTRA = _extra

; Initialize the superclass.
IF (self -> IDLVisualization::Init(/REGISTER_PROPERTIES, $
  TYPE='ExampleVis', NAME='Example Visualization Type', $
```

```

        ICON='plot', _EXTRA = _extra) NE 1) THEN $
        RETURN, 0

; Register a parameter
self -> RegisterParameter, 'Y', DESCRIPTION='Y Plot Data', $
    /INPUT, TYPES='IDLVECTOR', /OPTARGET

; Add a plotting symbol object and aggregate its properties
; into the visualization.
self._oSymbol = OBJ_NEW('IDLitSymbol', PARENT = self)
self -> Aggregate, self._oSymbol

; Create an IDLitVisPlot object, setting its SYMBOL property to
; the symbol object we just created. Add the plot object to the
; visualization, and aggregate its properties.
self._oPlot = OBJ_NEW('IDLitGrPlot', /REGISTER_PROPERTIES, $
    SYMBOL = self._oSymbol -> GetSymbol())
self -> Add, self._oPlot, /AGGREGATE

; Register an example property that holds a string value.
self -> RegisterProperty, 'ExampleProperty', $
    /STRING, DESCRIPTION='An example property', $
    NAME='Example Property', SENSITIVE = 1

; Pass any extra keyword parameters through to the SetProperty
; method.
IF (N_ELEMENTS(_extra) GT 0) THEN $
    self -> ExampleVis::SetProperty, _EXTRA = _extra

; Return success
RETURN, 1

END

```

## Discussion

The `ExampleVis` class is based on the `IDLitVisualization` class (discussed in [“Subclassing from the IDLitVisualization Class”](#) on page 108). As a result, all of the standard features of an `iTool` visualization class are already present. We don’t define any keyword values to be handled explicitly in the `Init` method, but we do use the keyword inheritance mechanism to pass keyword values through to methods called within the `Init` method. The `ExampleVis` `Init` method does the following things:

1. Calls the `Init` method of the superclass, `IDLitVisualization`. We use the `REGISTER_PROPERTIES` keyword to ensure that all registrable properties of the superclass are exposed in the `ExampleVis` object’s property sheet. We also set the visualization type to be an “`ExampleVis`,” provide a `Name` for the object instance, and provide an icon. Finally, we use the `_EXTRA` keyword

inheritance mechanism to pass through any keywords provided when the `ExampleVis Init` method is called.

2. Registers an input parameter called `Y` that must be a vector. The `OPTARGET` keyword specifies that the `Y` parameter can be the target for `iTool` operations.
3. Creates a plotting symbol created from the `IDLitSymbol` class and aggregate its properties with the other `ExampleVis` properties.
4. Creates an `IDLitGrPlot` object that uses the `IDLitSymbol` object for its plotting symbols.
5. Registers an example property that holds a string value.
6. Passes any “extra” keyword properties through to the  `SetProperty` method.
7. Returns the integer 1, indicating successful initialization.

## Creating a Cleanup Method

The visualization class `Cleanup` method handles any cleanup required by the visualization object, and should do the following:

- destroy any objects created by the visualization that were not added to the graphics hierarchy with a call to the `Add` method
- call the superclass’ `Cleanup` method

Calling the superclass’ `cleanup` method will destroy any objects that were added to the graphics hierarchy.

See “[IDLitVisualization::Cleanup](#)” in the *IDL Reference Guide* manual for additional details.

### Example Cleanup Method

The following example code shows a very simple `Cleanup` method for the `ExampleVis` visualization type:

```
PRO ExampleVis::Cleanup

    ; Clean up the IDLitSymbol object we created.
    OBJ_DESTROY, self._oSymbol

    ; Call superclass Cleanup method
    self -> IDLitVisualization::Cleanup

END
```

## Discussion

The Cleanup method first destroys the IDLitSymbol object, which is not part of the graphics hierarchy, then calls the superclass Cleanup method to destroy the objects in the graphics hierarchy.

## Creating a GetProperty Method

The visualization class GetProperty method retrieves property values from the visualization object instance or from instance data of other associated objects. The method can retrieve the requested property value from the visualization object's instance data or by calling another class' GetProperty method.

### Note

---

Any property registered with a call to the RegisterProperty method must be listed as a keyword to the GetProperty method either of the visualization class or one of its superclasses.

---

See “IDLitVisualization::GetProperty” in the *IDL Reference Guide* manual for additional details.

## Example GetProperty Method

The following example code shows a very simple GetProperty method for the ExampleVis visualization type:

```
PRO ExampleVis::GetProperty, $
    EXAMPLEPROPERTY = exampleProperty, $
    _REF_EXTRA = _extra

    IF ARG_PRESENT(exampleProperty) THEN BEGIN
        exampleProperty = self._exampleproperty
    ENDIF

    ; get superclass properties
    IF (N_ELEMENTS(_extra) GT 0) THEN $
        self -> IDLitVisualization::GetProperty, _EXTRA = _extra

END
```

## Discussion

The GetProperty method first defines the keywords it will accept. There must be a keyword for each property of the visualization type. The keyword inheritance mechanism allows properties to be retrieved from the ExampleVis class' superclasses without knowing the names of the properties.

Using the ARG\_PRESENT function, the method checks for the presence of keywords in the call to the GetProperty method. If a keyword is detected, it retrieves the value of the associated property from the object's instance data. In this example, only one property (ExampleProperty) is specific to the ExampleVis object.

Finally, the method calls the superclass' GetProperty method, passing in all of the keywords stored in the \_EXTRA structure.

## Creating a SetProperty Method

The visualization class SetProperty method stores property values in the visualization object's instance data or in properties of associated objects. It sets the specified property value either by storing the value directly in the visualization object's instance data or by calling another class' SetProperty method.

### Note

Any property registered with a call to the RegisterProperty method must be listed as a keyword to the SetProperty method either of the visualization class or one of its superclasses.

See [“IDLitVisualization::SetProperty”](#) in the *IDL Reference Guide* manual for additional details.

## Example SetProperty Method

The following example code shows a very simple SetProperty method for the ExampleVis visualization type:

```
PRO ExampleVis::SetProperty, $
  EXAMPLEPROPERTY = exampleProperty, $
  _EXTRA = _extra

  IF (N_ELEMENTS(exampleProperty) GT 0) THEN BEGIN
    self._exampleProperty = exampleProperty
  ENDIF

  IF (N_ELEMENTS(_extra) GT 0) THEN $
    self -> IDLitVisualization::SetProperty, _EXTRA = _extra

END
```

### Discussion

The SetProperty method first defines the keywords it will accept. There must be a keyword for each property of the visualization type. The keyword inheritance



mechanism allows properties to be set on the `ExampleVis` class' superclasses without knowing the names of the properties.

Using the `N_ELEMENTS` function, we check to see whether a value was specified for each keyword. If a value is detected, we set the value of the associated property. In this example, only one property (`ExampleProperty`) is specific to the `ExampleVis` object. We set the value of the `ExampleProperty` directly in the `ExampleVis` object's instance data.

Finally, we call the superclass' `SetProperty` method, passing in all of the keywords stored in the `_EXTRA` structure.

## Creating an `OnDataChangeUpdate` Method

The visualization class `OnDataChangeUpdate` method takes care of updating the visualization when one or more of the data parameters used to create the visualization change their values. The tasks this method must perform are dependent on the type of visualization involved and the data parameter that changes. The general idea is that when the value of a data object changes, the `OnDataChangeUpdate` method for each visualization that uses that data is called. The `OnDataChangeUpdate` method then uses the `GetData` method to retrieve the changed data from the `IDLitData` object, inspects the data and manipulates it as necessary, and uses the `SetProperty` method to insert the new data values into the visualization object.

See “[IDLitParameter::OnDataChangeUpdate](#)” in the *IDL Reference Guide* manual and “[Data Update Mechanism](#)” on page 45 for additional details.

### Example `OnDataChangeUpdate` Method

The following example code shows a very simple `OnDataChangeUpdate` method for the `ExampleVis` visualization type:

```
PRO ExampleVis::OnDataChangeUpdate, oSubject, parmName

CASE STRUPCASE(parmName) OF

  '<PARAMETER SET>': BEGIN
    oParams = oSubject -> Get(/ALL, COUNT = nParam, $
      NAME = paramNames)
    FOR i = 0, nParam-1 DO BEGIN
      IF (paramNames[i] EQ '') THEN CONTINUE
      oData = oSubject -> GetByName(paramNames[i])
      IF (OBJ_VALID(oData)) THEN $
        self -> OnDataChangeUpdate, oData, paramNames[i]
    ENDFOR
  END
```

```

'Y': BEGIN
    success = oSubject -> GetData(data)
    nData = N_ELEMENTS(data)
    IF (nData GT 0) THEN BEGIN
        ; Set the min/max values.
        minn = MIN(data, MAX = maxx)
        self._oPlot -> SetProperty, DATAY = TEMPORARY(data), $
            MIN_VALUE = minn, MAX_VALUE = maxx
    ENDIF
END
ELSE: self -> ErrorMessage, 'Unknown parameter'
ENDCASE

END

```

## Discussion

The `OnDataChangeUpdate` method must accept two arguments: an object reference to the data object whose data has changed (`oSubject` in the previous example), and a string containing the name of the parameter associated with the data object (`parmName` in the example).

## Note

---

The string `<PARAMETER SET>` is a special case value for the second argument, used to indicate that the object reference is not a single data object but a parameter set. Calling `OnDataChangeUpdate` with a parameter set rather than a data item provides a simple way to update a group of data values in with a single statement; this can be very useful when creating the visualization for the first time.

---

We use a CASE statement to determine which parameter has been modified, and process the data as appropriate. We first handle the special case where the parameter has the value `<PARAMETER SET>` by looping through all of the parameters in the parameter set object, calling the `OnDataChangeUpdate` method again on each parameter.

Next, we handle the parameter (Y) by calling the `IDLitData::GetData` method on the data object reference stored in the `oSubject` argument. The second argument (the string `'IDLVECTOR'`) instructs the `GetData` method to retrieve only data of vector type. We use the `N_ELEMENTS` function to determine whether any data was returned. If data was returned, we determine the minimum and maximum values. Finally, we use the `SetProperty` method to insert the changed data (using the `TEMPORARY` function to avoid making a copy of the data) into the `DATAY` property of the `IDLitVisPlot` object stored in the visualization's `_oPlot` class structure field. Similarly, we insert the new minimum and maximum values into the `MIN_VALUE` and `MAX_VALUE` properties of the `IDLitVisPlot` object.

## Creating an OnDataDisconnect Method

The visualization class `OnDataDisconnect` method is called automatically when a data value has been disconnected from a parameter. A visualization class based on the `IDLitVisualization` class *must* implement this method in order for changes or additions to the data parameters to be updated automatically in the resulting visualizations. The general idea is that when a data item is disassociated from a visualization parameter, one or more properties of the visualization may need to be reset to reasonable default values. For example, in the case of a plot visualization, if the plotted data is disconnected, we want to reset the data ranges to their default values and hide the plot visualization.

See “[IDLitParameter::OnDataDisconnect](#)” in the *IDL Reference Guide* manual for additional details.

### Example OnDataDisconnect Method

```
PRO ExampleVis::OnDataDisconnect, ParmName

CASE ParmName OF
  'Y': BEGIN
    self._oPlot -> SetProperty, DATAX = [0,1], DATAY = [0,1]
    self._oPlot -> SetProperty, /HIDE
  END

  ELSE:
ENDCASE

END
```

### Discussion

The `OnDataDisconnect` method takes a single argument, which contains the uppercase name of the parameter that was disconnected. In the case of our `ExampleVis` visualization, we only need to handle the `Y` parameter. If the `Y` parameter is disconnected, we set the data ranges of the plot object to their default values (the range between 0 and 1), and hide the plot visualization using the `HIDE` property.

## Creating the Class Structure Definition

When any IDL object is created, IDL looks for an IDL class structure definition that specifies the instance data fields needed by an instance of the object, along with the data types of those fields. The object class structure must have been defined *before* any objects of the type are created. In practice, when the IDL `OBJ_NEW` function attempts to create an instance of a specified object class, it executes a procedure

named `ObjectClass__define` (where `ObjectClass` is the name of the object), which is expected to define an IDL structure variable with the correct name and structure fields. For additional information on how IDL creates object instances, see “[The Object Lifecycle](#)” in Chapter 22 of the *Building IDL Applications* manual.

## Subclassing from the IDLitVisualization Class

The IDLitVisualization class serves as a container for visualization objects displayed in an iTool. The class includes methods to handle changes to data and property values automatically; in almost all cases, new visualization types will be subclassed from the IDLitVisualization class. See “[IDLitVisualization](#)” in the *IDL Reference Guide* manual for details on the methods properties available to classes that subclass from IDLitVisualization.

## Example Class Structure Definition

The following is the class structure definition for the `ExampleVis` visualization class. This procedure should be the last procedure in a file named `examplevis__define.pro`.

```
PRO ExampleVis__Define

    struct = { ExampleVis,          $
              INHERITS IDLitVisualization, $
              _oPlot: OBJ_NEW(),      $
              _oSymbol: OBJ_NEW(),    $
              _exampleProperty: ' '   $
            }

END
```

## Discussion

The purpose of the structure definition routine is to define a named IDL structure with structure fields that will contain the visualization object instance data. The structure name should be the same as the visualization’s class name — in this case, `ExampleVis`.

Like many iTool visualizations, `ExampleVis` is created as a subclass of the IDLitVisualization class. Visualization classes that subclass from the IDLitVisualization class inherit all of the standard iTool visualization features, as described in “[Subclassing from the IDLitVisualization Class](#)” on page 108.

The `ExampleVis` visualization class instance data includes two graphics objects: an IDLitVisPlot object, to which a reference is stored in the `_oPlot` class structure field, and an IDLitVisSymbol object, to which a reference is stored in the `_oSymbol` class

structure field. Both graphics objects are defined in the class structure definitions as object instances, denoted by the presence of the `OBJ_NEW()` after the structure field name. Finally, instance data for a string property named `ExampleProperty` is stored in the `_exampleProperty` class structure field.

**Note**

---

This example is intended to demonstrate how simple it can be to create a new visualization class definition. While the class definition for a visualization class with significant extra functionality will likely define additional structure fields, and may inherit from other iTool classes, the basic principles are the same.

---

# Registering a Visualization Type

Before a visualization of a given type can be created by an iTool, the visualization type's class definition must be registered as being available to the iTool. Registering a visualization type with the iTool links the class definition file containing the actual IDL code that defines the visualization type with a simple string that names the type. Code that creates a visualization in an iTool uses the name string to specify which type of visualization should be created. In addition, some operations and manipulators will operate only on specific visualization types; these limits are also specified using the name string.

## Using IDLitool::RegisterVisualization

In most cases, you will register a visualization type with the iTool in the iTool's class Init method. Registration ensures that the visualization type is available when the iTool attempts to create a visualization. (See [“Creating a New iTool Class”](#) on page 69 for details on the iTool class Init method.)

To register a visualization, call the IDLitool::RegisterVisualization method:

```
self -> RegisterVisualization, Visualization_Type, $
      VisType_Class_Name
```

where *Visualization\_Type* is the string you will use when referring to the visualization type, and *VisType\_Class\_Name* is a string that specifies the name of the class file that contains the visualization type's definition.

---

**Note**

The file *VisType\_Class\_Name\_\_define.pro* must exist somewhere in IDL's path for the visualization type to be successfully registered.

---

See [“IDLitool::RegisterVisualization”](#) in the *IDL Reference Guide* manual for details.

## Specifying Useful Properties

You can set any property of the [IDLitVisualization](#) and [IDLitComponent](#) classes when registering a visualization. The following properties may be of particular interest:

**ICON**

A string value giving the name of an icon to be associated with this object. Typically, this property is the name of a bitmap file to be used when displaying the object in a tree view. See [“Icon Bitmaps”](#) on page 28 for details on where bitmap icon files are located.

**TYPE**

A string or an array of strings indicating the types of data that can be displayed by the visualization. iTools data types are described in [Chapter 3, “Data Management”](#). Set this property to a null string ( ' ' ) to specify that all types of data can be displayed.

# Unregistering a Visualization Type

If you are creating a new `iTool` from an existing `iTool` class, you may want to remove a visualization type registered with the existing class from your new tool. This can be useful if you have an `iTool` class that implements all of the functionality you need, but which registers a visualization type you don't want included in your `iTool`. Rather than recreating the `iTool` class to remove the visualization type, you could create your new `iTool` class in such a way that it inherits from the existing `iTool` class, but *unregisters* the unwanted visualization.

Unregister a visualization type by calling the `IDLitTool::UnregisterVisualization` method in the `Init` method of your `iTool` class:

```
self -> UnregisterVisualization, identifier
```

where *identifier* is the string name used when registering the visualization.

For example, suppose you are creating a new `iTool` that subclasses from the standard `iSurface` tool, which is defined by the `IDLitToolSurface` class. If you wanted your new tool to behave just like the `iSurface` tool, with the exception that it would not handle 2D plot visualizations, you could include the following method call in your `iTool`'s `Init` method:

```
self -> UnregisterVisualization, 'Plot'
```

## Finding the Identifier String

To find the string value used as the *identifier* parameter to the `UnregisterVisualization` method, you must inspect the class file that registers the visualization. In the case of our example, you would inspect the file `idlittoolsurface__define.pro` to find the following call to the `RegisterVisualization` method:

```
self -> RegisterVisualization, 'Plot', 'IDLitVisPlot', $  
      ICON = 'plot'
```

The first argument to the `RegisterVisualization` method (`'Plot'`) is the string name of the visualization type.



# Example: Image-Contour Visualization

This example creates a visualization type named `visImageContour` that displays an image and overlays it with a contour based on the image data.

## Class Definition File

The class definition for `visImageContour` consists of an `Init` method, an `OnDataChangeUpdate` method, and a class structure definition routine. Other important methods — `Cleanup`, `GetProperty`, and `SetProperty` — are handled by the superclass (`IDLitVisualization`).

As with all object class definition files, the class structure definition routine is the last routine in the file, and the file is given the same name as the class definition routine (with the suffix `.pro` appended).

## Init Method

The `Init` method is called when the `visImageContour` visualization is created.

```
FUNCTION visImageContour::Init, _EXTRA = _extra

    ; Initialize the superclass
    IF (~self -> IDLitVisualization::Init(NAME='visImageContour', $
        ICON = 'image', _EXTRA = _extra)) THEN RETURN, 0

    ; Register the parameters we are using FOR data
    self -> RegisterParameter, 'IMAGEPIXELS', $
        DESCRIPTION = 'Image Data', /INPUT, $
        TYPES = ['IDLIMAGEPIXELS', 'IDLARRAY2D'], /OPTARGET
    self -> RegisterParameter, 'PALETTE', $
        DESCRIPTION = 'Palette', /INPUT, /OPTIONAL, $
        TYPES = ['IDLPALETTE', 'IDLARRAY2D'], /OPTARGET

    ; Create objects and add to this Visualization
    self._oImage = OBJ_NEW('IDLitVisImage', _EXTRA = _extra)
    self -> Add, self._oImage, /AGGREGATE
    self._oContour = OBJ_NEW('IDLitVisContour', _EXTRA = _extra)
    self -> Add, self._oContour, /AGGREGATE

    ; Return success
    RETURN, 1

END
```

## Discussion

The first item in our class definition file is the `Init` method. The `Init` method's function signature is defined first, using the class name `visImageContour`. Note the use of the `_EXTRA` keyword inheritance mechanism; this allows any keywords specified in a call to the `Init` method to be passed through to routines that are called within the `Init` method even if we do not know the names of those keywords in advance.

First, we call the `Init` method of the superclass. In this case, we are creating a subclass of the `IDLitVisualization` class; this provides us with all of the standard `iTool` visualization methods automatically. Any “extra” keywords specified in the call to our `Init` method are passed to the `IDLitVisualization::Init` method via the keyword inheritance mechanism. If the call to the superclass `Init` method fails, we return immediately with a value of 0.

We register two parameters used by our visualization: `IMAGEPIXELS` and `PALETTE`. Both parameters are input parameters (meaning they are used to create the visualization), and both can be the target of an operation. The `IMAGEPIXELS` parameter can contain data of two `iTool` data types: `IDLIMAGEPIXELS` or `IDLARRAY2D`. When data are assigned to the visualization's parameter set, only data that matches one of these two types can be assigned to the `IMAGEPIXELS` parameter. Similarly, the `PALETTE` parameter can contain data of type `IDLPALETTE` or `IDLARRAY2D`.

Next, we create the two visualization components that make up the `visImageContour` visualization type: an `IDLitVisImage` object and an `IDLitVisContour` object. Each object is created by a call to the `OBJ_NEW` function; the newly-created object reference is placed in a field of the `visImageContour` object's instance data structure. The new visualization objects are then added to the `visImageContour` object using the `Add` method; the `AGGREGATE` keyword specifies that the properties of each of the component visualization objects will be displayed as properties of the `visImageContour` object itself.

Finally, we return 1, indicating a successful initialization.

## OnChangeUpdate Method

The `OnChangeUpdate` method is called whenever the data associated with the `visImageContour` visualization object changes. This may include the initial creation of the visualization, if data parameters are specified in the call to the `iTool` launch routine that creates the visualization.

```

PRO visImageContour::OnDataChangeUpdate, oSubject, parmName, $
    _REF_EXTRA = _extra

    ; Branch based on the value of the parmName string.
    CASE STRUPCASE(parmName) OF

        ; The method was called with a parameter set as the argument.
        '<PARAMETER SET>': BEGIN
            oParams = oSubject -> Get(/ALL, COUNT = nParam, $
                NAME = paramNames)
            FOR i = 0, nParam-1 DO BEGIN
                IF (paramNames[i] EQ '') THEN CONTINUE
                oData = oSubject -> GetByName(paramNames[i])
                IF (OBJ_VALID(oData)) THEN $
                    self -> OnDataChangeUpdate, oData, paramNames[i]
            ENDFOR
        END

        ; The method was called with an image array as the argument.
        'IMAGEPIXELS': BEGIN
            void = self._oImage -> SetData(oSubject, $
                PARAMETER_NAME = 'IMAGEPIXELS')
            void = self._oContour -> SetData(oSubject, $
                PARAMETER_NAME = 'Z')
            ; Make our contour appear at the top OF the surface.
            IF (oSubject -> GetData(zdata)) THEN $
                self._oContour -> SetProperty, ZVALUE = MAX(zdata)
        END

        ; The method was called with a palette as the argument.
        'PALETTE': BEGIN
            void = self._oImage -> SetData(oSubject, $
                PARAMETER_NAME = 'PALETTE')
            void = self._oContour -> SetData(oSubject, $
                PARAMETER_NAME = 'PALETTE')
        END

        ELSE: ; DO nothing

    ENDCASE

END

```

## Discussion

The `OnDataChangeUpdate` method accepts the two required arguments: an object reference to the data object whose data has changed (`oSubject`), and a string containing the name of the parameter associated with the data object (`parmName`).

We use a CASE statement to determine which parameter has been modified, and process the data as appropriate. We first handle the special case where the parameter has the value `<PARAMETER SET>` by looping through all of the parameters in the parameter set object, calling the `OnChangeUpdate` method again on each parameter.

We handle the `IMAGEPIXELS` parameter by calling the `IDLitParameter::SetData` method once on each of the two component visualizations, specifying that the input data object `oSubject` corresponds to the `IMAGEPIXELS` parameter of the `IDLitVisImage` object, and to the `Z` parameter of the `IDLitVisContour` object. We also set the `Z` value of the `IDLitVisContour` object using the maximum data value of the data contained in `oSubject`.

Finally, we handle the `PALETTE` parameter by calling the `SetData` method again, this time to set the `PALETTE` parameters of both the `IDLitVisImage` and `IDLitVisContour` objects.

## OnDataDisconnect Method

The `OnDataDisconnect` method is called automatically when a data value has been disconnected from a parameter.

```
PRO visImageContour::OnDataDisconnect, ParmName

CASE STRUPCASE(parmname) OF

    'IMAGEPIXELS': BEGIN
        self -> SetProperty, DATA = 0
        self._oImage -> SetProperty, /HIDE
        self._oContour -> SetProperty, /HIDE
    END

    'PALETTE': BEGIN
        self._oImage -> SetProperty, PALETTE = OBJ_NEW()
        self -> SetPropertyAttribute, 'PALETTE', SENSITIVE = 0
    END

    ELSE: ; DO nothing
ENDCASE

END
```

## Discussion

The `OnDataDisconnect` method takes a single argument, which contains the name of the parameter that was disconnected. In the case of our `visImageContour` visualization, we handle the `IMAGEPIXELS` and `PALETTE` parameters. For the

IMAGEPIXELS parameter, we set the DATA property of the parameter to 0, and hide both the image and the contour visualizations. For the PALETTE parameter, we set the PALETTE property of the image to a null object, and desensitize the property in the property sheet display.

## Class Definition

```
PRO visImageContour__Define
  struct = { visImageContour, $
    inherits IDLitVisualization, $
    _oContour: OBJ_NEW(), $
    _oImage: OBJ_NEW() $
  }
END
```

## Discussion

Our class definition routine creates an IDL structure variable with the name `visImageContour`, specifying that the structure inherits from the `IDLitVisualization` class. The structure has two instance data fields named `_oContour` and `_oImage`, which will contain object references to the `IDLitVisImage` and `IDLitVisContour` objects that make up the `visImageContour` visualization.





# Chapter 7: Creating an Operation

This chapter describes the process of creating an iTool operation.

---

Overview .....	120	Creating a New Generalized Operation ..	138
Predefined iTool Operations .....	122	Registering an Operation .....	153
Operations and the Undo/Redo System ...	123	Unregistering an Operation .....	155
Creating a New Data-Centric Operation ..	125	Example: Data Resample Operation ....	156

# Overview

An *operation* is an iTool component object class that can be used to modify selected data, change the way a visualization is displayed in the iTool window, or otherwise affect the state of the iTool. Some examples of iTool operations are:

- performing the IDL SMOOTH operation on selected data,
- rotating a selected visualization by a specified angle,
- displaying data statistics.

A number of standard operations are predefined and included in the IDL iTools package; if none of the predefined operations suits your needs, you can create your own operation by subclassing either from the base IDLitOperation class on which all of the predefined operations are based, from the IDLitDataOperation class, or from one of the predefined operations.

## The Operation Creation Process

To create a new iTool operation, you will do the following:

- Choose an iTool operation class on which your new operation will be based. In most cases, the operation will act on the data underlying a visualization; in these cases, you will base your new operation on the IDLitDataOperation class. If your operation will affect something other than data — the appearance of visualizations in the iTool window, or the value of some property — you will base your new class on the IDLitOperation class. Both classes provide support for the iTool undo/redo system, but operations that do not deal directly with data require additional code to properly allow for undoing and redoing the operations.
- Define the properties of the operation, and set default property values.
- If the new operation acts directly on data (that is, if it is based on the IDLitDataOperation class), provide an Execute method that performs the operation using the current property values. Similarly, if the new operation is more general and is based on the IDLitOperation class, provide a DoAction method.
- Optionally provide a DoExecuteUI method to display a user interface for operations that act directly on data.
- For generalized operations, provide UndoOperation and RedoOperation methods to undo and redo the operation. These methods may in turn require



that you provide methods to store values before and after the operation is executed.

- Override methods used to get or set properties, react to changes in the underlying data, and clean up, as necessary.

This chapter describes the process of creating new operations based on the `IDLitDataOperation` and `IDLitOperation` classes.

# Predefined iTool Operations

The iTool system distributed with IDL includes a number of pre-defined operations. You can include these operations in an iTool directly by registering the class with your iTool (as described in [“Registering an Operation”](#) on page 153). You can also create a new operation class based on one of the pre-defined classes.

## IDLitOpBytscl

Scales the values contained in a two-dimensional array into the range of 0-255

### Data Types Accepted

- IDLARRAY2D

## IDLitOpConvolution

Displays a dialog that allows the user to choose convolution settings, then calls the CONVOL function on the selected data using the specified parameters.

### Data Types Accepted

- IDLVECTOR, IDLARRAY2D, IDLIMAGE

## IDLitOpCurvefitting

Displays a dialog that allows the user to select a curve-fitting algorithm, then calls the appropriate IDL routine to perform the fit. The fitted curve is then created and inserted into the visualization as a new plot line.

### Data Types Accepted

- IDLVECTOR

## IDLitOpSmooth

Calls the SMOOTH function on the selected data. The smoothing window parameter can be set by the user via the property sheet interface of the Operations browser.

### Data Types Accepted

- IDLVECTOR, IDLARRAY2D

# Operations and the Undo/Redo System

The iTools system provides users with the ability to interactively undo and redo actions performed on visualizations or data items. As an iTool developer, you will need to provide some code to support the undo/redo feature; the amount of code required depends largely on the type of operation your operation class performs. The main dividing line is between data-centric operations that act directly on the data that underlies a visualization, and operations that act in a more generalized way, changing some value that may not be directly related to a data item. In most cases, operations that act directly on data are based on the `IDLitDataOperation` class, whereas operations that are more generalized are based on the `IDLitOperation` class.

## Data-Centric Operations

Undo/redo functionality is handled automatically for data-centric operations based on the `IDLitDataOperation` class. The following things happen when the user requests an operation:

- For each selected item, data that matches the type supported by the operation is extracted and passed to the operation's `Execute` method. The `Execute` method modifies the data *in place*. When the data changes, all visualizations that observe the data item are notified, and update accordingly.
- If the user undoes the operation, the original data values are restored. By default, the original values are cached before the `Execute` method is called, and undoing the operation simply retrieves the data values from the cache. If the `REVERSIBLE_OPERATION` property of the `IDLitDataOperation` object is set, however, the original values are not cached, and the `UnExecute` method is called when the user undoes the operation. The `UnExecute` method must exist and must reverse the action performed by the `Execute` method, restoring the data items to their original values. Using the `REVERSIBLE_OPERATION` property allows you to avoid caching the data set (which may be large) when the operation performed on the data is easily reversed by computation.
- If the user redoes the operation, the data values computed by the `Execute` method are restored. By default, the `Execute` method is simply called again. If the `EXPENSIVE_OPERATION` property of the `IDLitDataOperation` object is set, however, the computed values are cached after the `Execute` method is called, and redoing the operation simply restores the cached data values. Using the `EXPENSIVE_OPERATION` property allows you to avoid having to recompute a computationally-intensive operation each time the user undoes and then redoes the operation.

## Generalized Operations

To provide undo/redo functionality, generalized operations (those based on the `IDLitOperation` class) must provide methods that record the initial and final values of the item being modified, along with methods that use the recorded values to undo or redo the operation. The following things happen when the user requests an operation:

- The `DoAction` method creates an `IDLitCommandSet` object to hold the initial and final values.
- The `RecordInitialValues` method records the original values of the specified target objects. Values are stored as data items in `IDLitCommand` objects, which are in turn stored in the `IDLitCommandSet` object.
- The `RecordFinalValues` method retrieves the `IDLitCommand` objects created by the `RecordInitialValues` method from the `IDLitCommandSet` object, and records the new values of the target objects as additional items in those `IDLitCommand` objects.
- If the user undoes the operation, the `UndoOperation` method retrieves the `IDLitCommand` objects from the `IDLitCommandSet` object, selects the relevant data items from each, and restores the values.
- If the user redoes the operation, the `RedoOperation` method retrieves the `IDLitCommand` objects from the `IDLitCommandSet` object, selects the relevant data items from each, and restores the values.

# Creating a New Data-Centric Operation

iTool operations that act primarily on data are based on the `IDLitDataOperation` class. The class definition file for an `IDLitDataOperation` object must (at the least) provide methods to initialize the operation class, get and set property values, execute the operation, and define the operation class structure. Complex operations will likely provide additional methods.

## How an `IDLitDataOperation` Works

When an `IDLitDataOperation` is requested by a user, the following things occur:

1. As with any operation, execution commences when the `DoAction` method is called. When called, the `IDLitDataOperation` retrieves the currently-selected items. If nothing is selected, the operation returns.
2. For each selected item, the data objects of the parameters registered as “operation targets” are retrieved.
3. The data objects are queried for iTool data types that match the types supported by the `IDLitDataOperation`.

For each data object that includes data of an iTool data type supported by the `IDLitDataOperation`, the following things occur:

1. The data from the data object is retrieved.
2. If the `IDLitDataOperation` does not have the `REVERSIBLE_OPERATION` property set, a copy of the data is created and placed into the undo-redo command set.
3. The `Execute` method is called, with the retrieved data as its argument.
4. If the `Execute` method succeeds and the `IDLitDataOperation` has the `EXPENSIVE_OPERATION` property set, a copy of the results is placed into the undo-redo command set.
5. The result of the `IDLitDataOperation` is placed in the data object. This action will cause all visualization items that use the data object to update when the operation is completed.

Once all selected data items have been processed, the undo-redo command set is placed into the system undo-redo buffer for later use.

## Creating an IDLitDataOperation

The process of creating an IDLitDataOperation is outlined in the following sections:

- “[Creating an Init Method](#)” on page 126
- “[Creating a Cleanup Method](#)” on page 130
- “[Creating an Execute Method](#)” on page 131
- “[Creating a DoExecuteUI Method](#)” on page 132
- “[Creating a GetProperty Method](#)” on page 133
- “[Creating a SetProperty Method](#)” on page 134
- “[Creating an UndoExecute Method](#)” on page 135
- “[Creating the Class Structure Definition](#)” on page 136

## Creating an Init Method

The operation class Init method handles any initialization required by the operation object, and should do the following:

- define the Init function method
- call the Init methods of any superclasses
- register any properties of the operation, and set property attributes as necessary
- perform other initialization steps as necessary
- return the value 1 if the initialization steps are successful, or 0 otherwise

### Definition of the Init Function

Begin by defining the argument and keyword list for your Init method. The argument and keyword list defines positional parameters (arguments) accepted by your method, defines any keywords that will be handled directly by your method, and specifies whether keywords not explicitly handled by your method will be passed through to other routines called by your method via IDL’s keyword inheritance mechanism.

---

**Note**

Because iTool operations are invoked by the user’s interactive choice of an item from a menu, they generally do not accept any keywords of their own.

---

The function signature of an Init method for an operation generally looks something like this:

```
FUNCTION MyOperation::Init, _EXTRA = _extra
```

where *MyOperation* is the name of your operation class.

Use keyword inheritance (the `_EXTRA` keyword) to pass keyword parameters through to any called routines as necessary. (See “[Keyword Inheritance](#)” in Chapter 4 of the *Building IDL Applications* manual for details on IDL’s keyword inheritance mechanism.)

## Superclass Initialization

The operation class Init method should call the Init method of any required superclass. For example, if your operation class is based on an existing operation, you would call that operation’s Init method:

```
success = self -> SomeOperationClass::Init(_EXTRA = _extra)
```

where *SomeOperationClass* is the class definition file for the operation on which your new operation is based. The variable `success` contains a 1 if the initialization was successful.

### Note

---

Your operation class may have multiple superclasses. In general, each superclass’ Init method should be invoked by your class’ Init method.

---

## Error Checking

Rather than simply calling the superclass Init method, it is a good idea to check whether the call to the superclass Init method succeeded. The following statement checks the value returned by the superclass Init method; if the returned value is 0 (indicating failure), the current Init method also immediately returns with a value of 0:

```
IF ( self -> SomeOperationClass::Init() EQ 0 ) THEN RETURN, 0
```

This convention is used in all operation classes included with IDL. RSI strongly suggests that you include similar checks in your own class definition files.

## Keywords to the Init Method

Properties of the operation class can be set in the Init method by specifying the property names and values as IDL keyword-value pairs. In addition to any keywords implemented directly in the Init method of the superclass on which you base your class, the properties of the `IDLitOperation` class and the `IDLitComponent` class are

available to any operation class. See “[IDLitOperation Properties](#)” and “[IDLitComponent Properties](#)” in the *IDL Reference Guide* manual.

Use keyword inheritance (the `_EXTRA` keyword) to pass keyword parameters through to the superclass as necessary. (See “[Keyword Inheritance](#)” in Chapter 4 of the *Building IDL Applications* manual for details on IDL’s keyword inheritance mechanism.)

## Standard Base Class

While you can create your new operation class from any existing operation class, in many cases, data-centric operation classes you create will be subclassed directly from the base class `IDLitDataOperation`:

```
IF (self -> IDLitDataOperation::Init(_EXTRA = _extra) EQ 0) $
    THEN RETURN, 0
```

The `IDLitDataOperation` class provides the base `iTool` functionality used in the data-centric operation classes created by RSI. See “[Subclassing from the IDLitDataOperation Class](#)” on page 136 for details.

## Return Value

If all of the routines and methods used in the `Init` method execute successfully, it should indicate successful initialization by returning 1. Other operation classes that subclass from your operation class may check this return value, as your routine should check the value returned by any superclass `Init` methods called.

## Registering Properties

Operations can register properties with the `iTool`; registered properties show up in the property sheet interface, and can be modified interactively by users. The `iTool` property interface is described in detail in [Chapter 4, “Property Management”](#).

Register a property by calling the `RegisterProperty` method of the `IDLitComponent` class:

```
self -> RegisterProperty, PropertyIdentifier [, TypeCode] $
    [, ATTRIBUTE = value]
```

where *PropertyIdentifier* is a string that uniquely identifies the property, *TypeCode* is an integer between 0 and 9 specifying the property data type, and *ATTRIBUTE* is a property attribute. See “[Registering Properties](#)” on page 54 for details.



## Setting Property Attributes

If a property has already been registered, perhaps by a superclass of your operation class, you can change the registered attribute values using the `SetPropertyAttribute` method of the `IDLitComponent` class:

```
self -> SetPropertyAttribute, Identifier
```

where *Identifier* is the name of the keyword to the `GetProperty` and `SetProperty` methods used to retrieve or change the value of this property. (The *Identifier* is specified in the call to `RegisterProperty` either via the *PropertyName* argument or the `IDENTIFIER` keyword.) See “[Property Attributes](#)” on page 58 for additional details.

## Example Init Method

The following example code shows a very simple `Init` method for an operation named `ExampleDataOp`. This function would be included (along with the class structure definition routine and any other methods defined by the class) in a file named `exampledataop__define.pro`.

```
FUNCTION ExampleDataOp::Init, _EXTRA = _extra

; Initialize the superclass.
IF (self -> IDLitDataOperation::Init(TYPES=['IDLIMAGE'], $
    NAME='Example Data Operation', ICON='sum', $
    _EXTRA = _extra) NE 1) THEN $
    RETURN, 0

; Register a property that holds a byte value.
self -> RegisterProperty, 'ByteTop', $
    DESCRIPTION='An example property', $
    NAME='Byte Threshold', SENSITIVE = 1

; Return success
RETURN, 1

END
```

## Discussion

The `ExampleDataOp` class is based on the `IDLitDataOperation` class (discussed in “[Subclassing from the IDLitDataOperation Class](#)” on page 136). As a result, all of the standard features of an `iTool` data operation are already present. We don’t define any keyword values to be handled explicitly in the `Init` method, but we do use the keyword inheritance mechanism to pass keyword values through to methods called within the `Init` method. The `ExampleDataOp` `Init` method does the following things:

1. Calls the Init method of the superclass, IDLitDataOperation. We use the TYPES keyword to specify that our operation works on data that has the iTool data type 'IDLIMAGE', provide a name for the object instance, and provide an icon. Finally, we use the \_EXTRA keyword inheritance mechanism to pass through any keywords provided when the ExampleDataOp Init method is called.
2. Registers a property that holds a byte value.
3. Returns the integer 1, indicating successful initialization.

## Creating a Cleanup Method

The operation class Cleanup method handles any cleanup required by the operation object, and should do the following:

- destroy any pointers or objects created by the operation
- call the superclass' Cleanup method

Calling the superclass' cleanup method will destroy any objects created when the superclass was initialized.

---

### Note

If your operation class is based on the IDLitDataOperation class, and does not create any pointers or objects of its own, the Cleanup method is not strictly required. It is always safest, however, to create a Cleanup method that calls the superclass' Cleanup method.

---

See “IDLitDataOperation::Cleanup” in the *IDL Reference Guide* manual for additional details.

## Example Cleanup Method

The following example code shows a very simple Cleanup method for the ExampleDataOp operation:

```
PRO ExampleDataOp::Cleanup

    ;; Cleanup superclass
    self -> IDLitDataOperation::Cleanup

END
```

## Discussion

Since our operation's instance data does not include any pointers or object references, the Cleanup method simply calls the superclass Cleanup method.

## Creating an Execute Method

The operation class Execute method does the computational work of a data-centric operation; it is called automatically when the iTool user requests an operation based on the IDLitDataOperation class. The Execute method must accept a single argument that contains the *raw data* associated with an item selected by the user.

The fact that the raw data is passed to the execute method means that the Execute method itself does not need to “unpack” a data object before performing the operations, allowing rapid and simple operation execution. For example, if the operation expects data of the iTools data type IDLARRAY2D, the iTool system will include the selected two-dimensional array as the *Data* argument.

The actual processing performed by the Execute method depends entirely on the operation.

## Example Execute Method

The following example code shows a simple Execute method for the ExampleDataOp operation, which will invert the values of the supplied data. Since our ExampleDataOp operation works on image data, this means the operation has the effect of producing the negative image.

```
FUNCTION ExampleDataOp::Execute, data

    ; If byte data then offsets are 0 and 255, otherwise
    ; use data minimum and maximum.
    offsetMax = (SIZE(data, /TYPE) eq 1) ? 255b : MAX(data)
    offsetMin = (SIZE(data, /TYPE) eq 1) ? 0b : MIN(data)
    data = offsetMax - TEMPORARY(data) + offsetMin
    RETURN, 1

END
```

## Discussion

When our ExampleDataOp operation is invoked by a user, the iTool system automatically checks to see which items are selected in the visualization window. For each selection, the iTool system extracts any data of type IDLIMAGE and passes that data to the Execute method as an IDL array. Our Execute method then finds the minium and maximum values, and inverts the data values.

## Creating a DoExecuteUI Method

Suppose we want to collect some information from the user before executing our operation. If the operation class sets the `SHOW_EXECUTION_UI` property, the `iTool` system will call the `DoExecuteUI` method before calling the `Execute` method. The `DoExecuteUI` method is responsible for displaying a user interface that collects the appropriate information and storing that information in properties of the operation object.

---

### Note

`iTools` provided with IDL that need to collect user input in this manner use the *UI service* mechanism, described in [Chapter 10, “iTool User Interface Architecture”](#). While it is possible for the `DoExecuteUI` method to perform all the necessary functions directly, using a UI service is the preferred method.

---

### Example DoExecuteUI Method

The following example code shows a simple `DoExecuteUI` method for the `ExampleDataOp` operation. This method relies on a UI service named `'ExampleDataOp'` being registered with the current `iTool`.

```
FUNCTION ExampleDataOp::DoExecuteUI

    oTool = self -> GetTool()
    IF (oTool EQ OBJ_NEW()) THEN RETURN, 0

    RETURN, oTool -> DoUIService('ExampleDataOp', self)

END
```

### Discussion

If the `SHOW_EXECUTION_UI` property is set on our `ExampleDataOp` operation object, the `DoExecuteUI` method is called automatically when the user invokes the operation. This method does the following:

1. Retrieve a reference to the current `iTool` object using the `GetTool` method of the `IDLitIMessaging` class. (`IDLitIMessaging` is a superclass of `IDLitOperation`, and thus of `IDLitDataOperation`.)
2. If the retrieved `iTool` object reference is a null object reference, no data about the current tool is available, so we return immediately without calling the UI service.

3. Call the ExampleDataOp UI service. Since our ExampleDataOp operation has only one property of its own (ByteTop), the ExampleDataOp UI presumably allows the user to set this value. See [Chapter 12, “Creating a User Interface Service”](#) for discussion of UI services.

## Creating a GetProperty Method

The operation class GetProperty method retrieves property values from the operation object instance or from instance data of other associated objects. It should retrieve the requested property value, either from the operation object’s instance data or by calling another class’ GetProperty method.

### Note

Any property registered with a call to the RegisterProperty method must be listed as a keyword to the GetProperty method either of the operation class or one of its superclasses.

See “IDLitDataOperation::GetProperty” in the *IDL Reference Guide* manual for additional details.

## Example GetProperty Method

The following example code shows a very simple GetProperty method for the ExampleDataOp operation:

```
PRO ExampleDataOp::GetProperty, $
    BYTETOP = byteTop, _REF_EXTRA = _extra

    IF ARG_PRESENT(byteTop) THEN BEGIN
        byteTop = self._byteTop
    ENDEF

    ; get superclass properties
    IF (N_ELEMENTS(_extra) GT 0) THEN $
        self -> IDLitDataOperation::GetProperty, _EXTRA = _extra

END
```

### Discussion

The GetProperty method first defines the keywords it will accept. There must be a keyword for each property of the operation type. The keyword inheritance mechanism allows properties to be retrieved from the ExampleDataOp class’ superclasses without knowing the names of the properties.

Using the ARG\_PRESENT function, we check for the presence of keywords in the call to the SetProperty method. If a keyword is detected, we retrieve the value of the associated property. In this example, only one property (ByteTop) is specific to the ExampleDataOp object. We retrieve the value of the ByteTop property directly from the ExampleDataOp object's instance data.

Finally, we call the superclass' SetProperty method, passing in all of the keywords stored in the \_EXTRA structure.

## Creating a SetProperty Method

The operation class SetProperty method stores property values in the operation object's instance data or in properties of associated objects. It should set the specified property value, either by storing the value directly in the operation object's instance data or by calling another class' SetProperty method.

### Note

Any property registered with a call to the RegisterProperty method must be listed as a keyword to the SetProperty method either of the operation class or one of its superclasses.

See “IDLitDataOperation::SetProperty” in the *IDL Reference Guide* manual for additional details.

## Example SetProperty Method

The following example code shows a very simple SetProperty method for the ExampleDataOp operation:

```
PRO ExampleDataOp::SetProperty, BYTETOP = byteTop, $
    _REF_EXTRA = _extra

    If (N_ELEMENTS(byteTop) GT 0) THEN BEGIN
        self._byteTop = byteTop
    ENDIF

    IF (N_ELEMENTS(_extra) GT 0) THEN $
        self -> IDLitDataOperation::SetProperty, _EXTRA = _extra

END
```

### Discussion

The SetProperty method first defines the keywords it will accept. There must be a keyword for each property of the operation. The keyword inheritance mechanism

allows properties to be set on the `ExampleDataOp` class' superclasses without knowing the names of the properties.

Using the `N_ELEMENTS` function, we check to see whether a value was specified for each keyword. If a value is detected, we set the value of the associated property. In this example, only one property (`ByteTop`) is specific to the `ExampleDataOp` object. We set the value of the `ExampleProperty` directly in the `ExampleDataOp` object's instance data.

Finally, we call the superclass' `SetProperty` method, passing in all of the keywords stored in the `_EXTRA` structure.

## Creating an UndoExecute Method

The operation class' `UndoExecute` method is called when the user undoes an invocation of the operation and the `REVERSIBLE_OPERATION` property is set on the operation object. (See [“Operations and the Undo/Redo System”](#) on page 123 for details on how undo and redo are handled in different situations.) The `UndoExecute` method must reverse the effect of the `Execute` method.

The actual processing performed by the `UndoExecute` method depends entirely on the operation.

### Example UndoExecute Method

The following example code shows a simple `UndoExecute` method for the `ExampleDataOp` operation, which reverses the operation of the `Execute` method.

```
FUNCTION ExampleDataOp::UndoExecute, data

    ; If byte data then offsets are 0 and 255, otherwise
    ; use data minimum and maximum.
    offsetMax = (SIZE(data, /TYPE) eq 1) ? 255b : MAX(data)
    offsetMin = (SIZE(data, /TYPE) eq 1) ? 0b : MIN(data)
    data = offsetMax - TEMPORARY(data) + offsetMin
    RETURN, 1

END
```

### Discussion

When the user undoes an invocation of our `ExampleDataOp` operation, the `iTool` system supplies the data that were computed by the `Execute` method when the operation was invoked. Our `UndoExecute` method then reverses the original operation.

## Creating the Class Structure Definition

When any IDL object is created, IDL looks for an IDL class structure definition that specifies the instance data fields needed by an instance of the object, along with the data types of those fields. The object class structure must be defined *before* any objects of the type are created. In practice, when the IDL OBJ\_NEW function attempts to create an instance of a specified object class, it executes a procedure named *ObjectClass\_\_define* (where *ObjectClass* is the name of the object), which is expected to define an IDL structure variable with the correct name and structure fields. For additional information on how IDL creates object instances, see “[The Object Lifecycle](#)” in Chapter 22 of the *Building IDL Applications* manual.

### Subclassing from the IDLitDataOperation Class

The IDLitDataOperation class simplifies the creation of operations that act only on data (as opposed to acting on the visual representation of that data) by providing methods that automate much of the process of execution and storing undo/redo data. If your operation class modifies data, you will almost certainly subclass from IDLitDataOperation, or from another operation that subclasses from IDLitDataOperation. See “[IDLitDataOperation](#)” in the *IDL Reference Guide* manual for details on the methods and properties available to classes that subclass from IDLitDataOperation.

### Example Class Structure Definition

The following is the class structure definition for the ExampleDataOp operation class. This procedure should be the last procedure in a file named `exampledataop__define.pro`.

```
PRO ExampleDataOp__Define

    struct = { ExampleDataOp,      $
              INHERITS IDLitDataOperation, $
              _byteTop: 0B      $
            }

END
```

### Discussion

The purpose of the structure definition routine is to define a named IDL structure with structure fields that will contain the operation object instance data. The structure name should be the same as the operation’s class name — in this case, `ExampleDataOp`.



Like many iTool operations that act on data, `ExampleDataOp` is created as a subclass of the `IDLitDataOperation` class. Operation classes that subclass from `IDLitDataOperation` class inherit methods and properties that make it easy to perform operations that affect data in an iTool.

The `ExampleDataOp` Operation class instance data includes a single property; a byte value that is stored in the `_byteTop` class structure field.

**Note** 

---

This example is intended to demonstrate how simple it can be to create a new operation class definition. While the class definition for an operation class with significant extra functionality will likely define additional structure fields, and may inherit from other iTool classes, the basic principles are the same.

---

# Creating a New Generalized Operation

*Generalized operations* are iTool operations that are not limited to acting on data that underlies a visualization. Generalized operations are based on the `IDLitOperation` class. The class definition file for an `IDLitOperation` object must (at the least) provide methods to initialize the operation class, get and set property values, execute the operation, undo and redo the operation, and define the operation class structure. Complex operations will likely provide additional methods.

## How an `IDLitOperation` Works

When an `IDLitOperation` is requested by a user, the operation's `DoAction` method (which must be provided by the operation class' developer) is called. The `DoAction` method is responsible for doing the following:

1. Retrieving the currently selected items and determining which items the operation should be applied to.
2. Creating an `IDLitCommandSet` object to contain undo/redo information.
3. Recording the initial values of the selected objects in the `IDLitCommandSet` object, if necessary.
4. Performing the actions associated with the operation.
5. Recording the final values of the selected objects in the `IDLitCommandSet` object, if necessary.
6. Returning the `IDLitCommandSet` object.

## Creating an `IDLitOperation`

The process of creating an `IDLitDataOperation` is outlined in the following sections:

- [“Creating an Init Method”](#) on page 139
- [“Creating a Cleanup Method”](#) on page 142
- [“Creating a DoAction Method”](#) on page 143
- [“Creating a RecordInitialValues Method”](#) on page 146
- [“Creating a RecordFinalValues Method”](#) on page 147
- [“Creating a GetProperty Method”](#) on page 147
- [“Creating a SetProperty Method”](#) on page 148

- “[Creating an UndoOperation Method](#)” on page 149
- “[Creating a RedoOperation Method](#)” on page 150
- “[Creating the Class Structure Definition](#)” on page 151

## Creating an Init Method

The operation class Init method handles any initialization required by the operation object, and should do the following:

- define the Init function method
- call the Init methods of any superclasses
- register any properties of the operation, and set property attributes as necessary
- perform other initialization steps as necessary
- return the value 1 if the initialization steps are successful, or 0 otherwise

### Definition of the Init Function

Begin by defining the argument and keyword list for your Init method. The argument and keyword list defines positional parameters (arguments) accepted by your method, defines any keywords that will be handled directly by your method, and specifies whether keywords not explicitly handled by your method will be passed through to other routines called by your method via IDL’s keyword inheritance mechanism.

---

#### Note

Because iTool operations are invoked by the user’s interactive choice of an item from a menu, they generally do not accept any keywords of their own.

---

The function signature of an Init method for an operation generally looks something like this:

```
FUNCTION MyOperation::Init, _EXTRA = _extra
```

where *MyOperation* is the name of your operation class.

Use keyword inheritance (the `_EXTRA` keyword) to pass keyword parameters through to any called routines as necessary. (See “[Keyword Inheritance](#)” in Chapter 4 of the *Building IDL Applications* manual for details on IDL’s keyword inheritance mechanism.)

## Superclass Initialization

The operation class Init method should call the Init method of any required superclass. For example, if your operation class is based on an existing operation, you would call that operation's Init method:

```
success = self -> SomeOperationClass::Init(_EXTRA = _extra)
```

where *SomeOperationClass* is the class definition file for the operation on which your new operation is based. The variable `success` contains a 1 if the initialization was successful.

### Note

---

Your operation class may have multiple superclasses. In general, each superclass' Init method should be invoked by your class' Init method.

---

## Error Checking

Rather than simply calling the superclass Init method, it is a good idea to check whether the call to the superclass Init method succeeded. The following statement checks the value returned by the superclass Init method; if the returned value is 0 (indicating failure), the current Init method also immediately returns with a value of 0:

```
IF (self -> SomeOperationClass::Init() EQ 0) THEN RETURN, 0
```

This convention is used in all operation classes included with IDL. RSI strongly suggests that you include similar checks in your own class definition files.

## Keywords to the Init Method

Properties of the operation class can be set in the Init method by specifying the property names and values as IDL keyword-value pairs. In addition to any keywords implemented directly in the Init method of the superclass on which you base your class, the properties of the IDLitOperation class and the IDLitComponent class are available to any operation class. See “[IDLitOperation Properties](#)” and “[IDLitComponent Properties](#)” in the *IDL Reference Guide* manual.

Use keyword inheritance (the `_EXTRA` keyword) to pass keyword parameters through to the superclass as necessary. (See “[Keyword Inheritance](#)” in Chapter 4 of the *Building IDL Applications* manual for details on IDL's keyword inheritance mechanism.)

## Standard Base Class

While you can create your new operation class from any existing operation class, in many cases, operations that do not act directly on the data that underlies a visualization will be subclassed directly from the base class `IDLitOperation`:

```
IF (self -> IDLitOperation::Init(_EXTRA = _extra) EQ 0) $
    THEN RETURN, 0
```

The `IDLitOperation` class provides the base iTTool functionality used in all operation classes created by RSI. See “[Subclassing from the IDLitOperation Class](#)” on page 152 for details.

## Return Value

If all of the routines and methods used in the `Init` method execute successfully, it should indicate successful initialization by returning 1. Other operation classes that subclass from your operation class may check this return value, as your routine should check the value returned by any superclass `Init` methods called.

## Registering Properties

Operations can register properties with the iTTool; registered properties show up in the property sheet interface, and can be modified interactively by users. The iTTool property interface is described in detail in [Chapter 4, “Property Management”](#).

Register a property by calling the `RegisterProperty` method of the `IDLitComponent` class:

```
self -> RegisterProperty, PropertyIdentifier [, TypeCode] $
    [, ATTRIBUTE = value]
```

where *PropertyIdentifier* is a string that uniquely identifies the property, *TypeCode* is an integer between 0 and 9 specifying the property data type, and *ATTRIBUTE* is a property attribute. See “[Registering Properties](#)” on page 54 for details.

## Setting Property Attributes

If a property has already been registered, perhaps by a superclass of your operation class, you can change the registered attribute values using the `SetPropertyAttribute` method of the `IDLitComponent` class:

```
self -> SetPropertyAttribute, Identifier
```

where *Identifier* is the name of the keyword to the `GetProperty` and `SetProperty` methods used to retrieve or change the value of this property. (The *Identifier* is specified in the call to `RegisterProperty` either via the *PropertyName* argument or the `IDENTIFIER` keyword.) See “[Property Attributes](#)” on page 58 for additional details.

## Example Init Method

The following example code shows a very simple Init method for an operation named `ExampleOp`. This function would be included (along with the class structure definition routine and any other methods defined by the class) in a file named `exampleop__define.pro`.

```
FUNCTION ExampleOp::Init, _EXTRA = _extra

; Initialize the superclass.
IF (self -> IDLitOperation::Init(TYPES=['IDLARRAY2D'], $
    NAME='Example Operation', ICON='generic_op', $
    _EXTRA = _extra) NE 1) THEN $
    RETURN, 0

; Return success
RETURN, 1

END
```

### Discussion

The `ExampleOp` class is based on the `IDLitOperation` class (discussed in [“Subclassing from the IDLitOperation Class”](#) on page 152). As a result, all of the standard features of an iTool operation are already present. We don’t define any keyword values to be handled explicitly in the Init method, but we do use the keyword inheritance mechanism to pass keyword values through to methods called within the Init method. The `ExampleOp` Init method does the following things:

1. Calls the Init method of the superclass, `IDLitOperation`. We use the `TYPES` keyword to specify that our operation works on data that has the iTool data type `'IDLARRAY2D'`, provide a Name for the object instance, and provide an icon. Finally, we use the `_EXTRA` keyword inheritance mechanism to pass through any keywords provided when the `ExampleOp` Init method is called.
2. Returns the integer 1, indicating successful initialization.

## Creating a Cleanup Method

The operation class Cleanup method handles any cleanup required by the operation object, and should do the following:

- destroy any pointers or objects created by the operation
- call the superclass’ Cleanup method

Calling the superclass' cleanup method will destroy any objects created when the superclass was initialized.

**Note**

If your operation class is based on the `IDLitOperation` class, and does not create any pointers or objects of its own, the `Cleanup` method is not strictly required. It is always safest, however, to create a `Cleanup` method that calls the superclass' `Cleanup` method.

See “[IDLitOperation::Cleanup](#)” in the *IDL Reference Guide* manual for additional details.

## Example Cleanup Method

The following example code shows a very simple `Cleanup` method for the `ExampleOp` operation:

```
PRO ExampleOp::Cleanup

    ;; Cleanup superclass
    self -> IDLitDataOperation::Cleanup

END
```

**Discussion**

Since our operation does not have any instance data of its own, the `Cleanup` method simply calls the superclass `Cleanup` method.

## Creating a DoAction Method

The operation class `DoAction` method is called by the *iTool* system when an operation is requested by the user. (Note that data-centric operations do not need to implement the `DoAction` method because it is implemented by the `IDLitDataOperation` class itself.) The `DoAction` method is responsible for the following:

- determining which objects the operation should be applied to (generally, but not always, the objects that are selected when the operation is invoked)
- retrieving the data from the selected objects
- creating an `IDLitCommandSet` object that will contain undo/redo data
- saving the state of the selected objects before the actions associated with the operation are performed in the command set object

- performing the requested actions on the selected objects
- saving the state of the selected objects after the actions associated with the operation are performed in the command set object
- returning the command set object

---

**Note**

If your operation changes the values of its own registered properties (as the result of user interaction with a dialog or other interface element called by `DoUIService`, for example), be sure to call the `RecordInitialValues` and `RecordFinalValues` methods. This ensures that changes made through the dialog are placed in the undo-redo transaction buffer.

---

## Example DoAction Method

The following example code shows a simple `DoAction` method for the `ExampleOp` operation. This operation retrieves the `STYLE` property of any selected `IDLitVisSurface` objects and increments its value by 1. Repeated invocations of this operation would cause the selected surfaces to loop through the seven available surface styles.

```
FUNCTION ExampleOp::DoAction, oTool

    ; Make sure we have a valid iTool object.
    IF ~ OBJ_VALID(oTool) THEN RETURN, OBJ_NEW()

    ; Get the selected objects
    oTargets = oTool -> GetSelectedItems()

    ; Select only IDLitVisSurface objects. If there are
    ; no surface objects selected, return a null object.
    surfaces = OBJ_NEW()
    FOR i = 0, N_ELEMENTS(oTargets)-1 DO BEGIN
        IF (OBJ_ISA(oTargets[i], 'IDLitVisSurface')) THEN BEGIN
            surfaces = OBJ_VALID(surfaces) ? $
                [surfaces, oTargets[i]] : oTargets[i]
        ENDIF
    ENDFOR

    IF (~OBJ_VALID(surfaces)) THEN RETURN, OBJ_NEW()

    ; Create a command set:
    oCmdSet = self -> IDLitOperation::DoAction(oTool)

    ; Record the initial values
    IF (~ self -> RecordInitialValues(oCmdSet, surfaces, '')) THEN $
```



```

        BEGIN
        OBJ_DESTROY, oCmdSet
        RETURN, OBJ_NEW()
    ENDIF

    ; Increment the style index for each surface.
    FOR i = 0, N_ELEMENTS(surfaces)-1 DO BEGIN
        ; Retrieve the current surface style and increment it
        surfaces[i] -> GetProperty, STYLE = styleIndex
        IF styleIndex eq 6 THEN BEGIN
            styleIndex = 0
        ENDIF ELSE BEGIN
            styleIndex += 1
        ENDELSE

        ; Set the new surface style
        oTargets[i] -> SetProperty, STYLE = styleIndex
    ENDFOR

    oTool->RefreshCurrentWindow

    ;; Record the final values
    result = self -> RecordFinalValues(oCmdSet, surfaces, '')

    RETURN, oCmdSet

END

```

## Discussion

The ExampleOp operation DoAction method does the following things:

1. Checks the validity of the iTool object passed to the DoAction method.
2. Retrieves the list of selected objects from the iTool object.
3. Filters out any selected objects that are not IDLitVisSurface objects.
4. Calls the superclass DoAction method to create an IDLitCommandSet object.
5. Calls the RecordInitialValues method to record the relevant values in the command set object before the operation is performed.
6. Loops through the list of IDLitVisSurface objects and increments the STYLE property of each by 1.
7. Calls the RecordFinalValues method to record the relevant values in the command set object after the operation has been performed.
8. Returns the command set object.

## Creating a RecordInitialValues Method

The operation class RecordInitialValues method is responsible for recording the appropriate “before” values from the specified objects in the provided IDLitCommandSet object. The values recorded depend entirely on the operation being performed.

### Example RecordInitialValues Method

The following example code shows a simple RecordInitialValues method for the ExampleOp operation. An IDLitCommand object is created for each of the target objects, and the value of the STYLE property of each object is recorded as an Item in the command object.

```
FUNCTION ExampleOp::RecordInitialValues, oCmdSet, oTargets, idProp

; Loop through the target objects and record the value of the
; STYLE property.
FOR i = 0, N_ELEMENTS(oTargets)-1 DO BEGIN
; Create a command object to store the values.
oCmd = OBJ_NEW('IDLitCommand', $
TARGET_IDENTIFIER = oTargets[i] -> GetFullIdentifier())
; Get the value of the STYLE property
oTargets[i] -> GetProperty, STYLE = styleIndex
; Add the value to the command object
void = oCmd -> AddItem('OLD_STYLE', styleIndex)
; Add the command object to the command set
oCmdSet -> Add, oCmd
ENDFOR

RETURN, 1

END
```

### Discussion

The ExampleOp operation RecordInitialValues method simply loops through the supplied list of target objects, creating a new IDLitCommand object for each. We set the TARGET\_IDENTIFIER property for each command object. Next, we retrieve the value of the STYLE property for each target object and add it to the command object as an Item. Finally, we add each command object to the supplied IDLitCommandSet object.

## Creating a RecordFinalValues Method

The operation class RecordFinalValues method is responsible for recording the appropriate “after” values from the specified objects in the provided IDLitCommandSet object. The values recorded depend entirely on the operation being performed.

### Example RecordFinalValues Method

The following example code shows a simple RecordFinalValues method for the ExampleOp operation. The new value of the STYLE property of each target object is recorded in the appropriate IDLitCommand object retrieved from the command set.

```
FUNCTION ExampleOp::RecordFinalValues, oCmdSet, oTargets, idProp

    ; Loop through the target objects and record the value of the
    ; STYLE property.
    FOR i = 0, N_ELEMENTS(oTargets)-1 DO BEGIN
        ; Retrieve the appropriate command object from the
        ; command set.
        oCmd = oCmdSet -> Get(POSITION = i)
        ; Get the value of the STYLE property
        oTargets[i] -> GetProperty, STYLE = styleIndex
        ; Add the value to the command object
        void = oCmd -> AddItem('NEW_STYLE', styleIndex)
        ; Add the command object to the command set
        oCmdSet -> Add, oCmd
    ENDFOR

    RETURN, 1

END
```

### Discussion

The ExampleOp operation RecordFinalValues method simply loops through the supplied list of target objects, recording the new value for the STYLE property in the IDLitCommand object associated with each target.

## Creating a GetProperty Method

The operation class GetProperty method retrieves property values from the operation object instance or from instance data of other associated objects. It should retrieve the requested property value, either from the operation object’s instance data or by calling another class’ GetProperty method.

**Note**

Any property registered with a call to the RegisterProperty method must be listed as a keyword to the GetProperty method either of the operation class or one of its superclasses.

See “IDLitOperation::GetProperty” in the *IDL Reference Guide* manual for additional details.

**Example GetProperty Method**

The following example code shows a very simple GetProperty method for the ExampleOp operation:

```
PRO ExampleOp::GetProperty, _REF_EXTRA = _extra

    ; get superclass properties
    IF (N_ELEMENTS(_extra) GT 0) THEN $
        self -> IDLitOperation::GetProperty, _EXTRA = _extra

END
```

**Discussion**

The GetProperty method first defines the keywords it will accept. There must be a keyword for each property of the operation type. The keyword inheritance mechanism allows properties to be retrieved from the ExampleOp class’ superclasses without knowing the names of the properties.

In this example, there are no properties specific to the ExampleOp object, so we simply call the superclass’ GetProperty method, passing in all of the keywords stored in the \_EXTRA structure.

**Creating a SetProperty Method**

The operation class SetProperty method stores property values in the operation object’s instance data or in properties of associated objects. It should set the specified property value, either by storing the value directly in the operation object’s instance data or by calling another class’ SetProperty method.

**Note**

Any property registered with a call to the RegisterProperty method must be listed as a keyword to the SetProperty method either of the operation class or one of its superclasses.

See “[IDLitOperation::SetProperty](#)” in the *IDL Reference Guide* manual for additional details.

## Example SetProperty Method

The following example code shows a very simple SetProperty method for the ExampleOp operation:

```
PRO ExampleOp::SetProperty, _EXTRA = _extra

    IF (N_ELEMENTS(_extra) GT 0) THEN $
        self -> IDLitOperation::SetProperty, _EXTRA = _extra

    END
```

### Discussion

The SetProperty method first defines the keywords it will accept. There must be a keyword for each property of the operation. The keyword inheritance mechanism allows properties to be set on the ExampleOp class’ superclasses without knowing the names of the properties.

In this example, there are no properties specific to the ExampleOp object, so we simply use the N\_ELEMENTS function to check whether the \_EXTRA structure contains any elements. If it does, we call the superclass’ SetProperty method, passing in all of the keywords stored in the \_EXTRA structure.

## Creating an UndoOperation Method

The operation class UndoOperation method is called when the user undoes the operation by selecting “Undo” from a menu or toolbar.

### Example UndoOperation Method

The following example code shows a very simple UndoOperation method for the ExampleOp operation:

```
FUNCTION ExampleOp::UndoOperation, oCommandSet

    ; Retrieve the IDLitCommand objects stored in the
    ; command set object.
    oCmds = oCommandSet -> Get(/ALL, COUNT = nObjs)

    ; Get a reference to the iTool object.
    oTool = self -> GetTool()

    ; Loop through the IDLitCommand objects and restore the
```

```

; original values.
FOR i = 0, nObjs-1 DO BEGIN
    oCmds[i] -> GetProperty, TARGET_IDENTIFIER = idTarget
    oTarget = oTool -> GetByIdentifier(idTarget)
    ; Get the old value
    IF (oCmds[i] -> GetItem('OLD_STYLE', styleIndex) EQ 1) THEN $
        oTarget -> SetProperty, STYLE = styleIndex
    ENDFOR
END

```

## Discussion

The UndoOperation method does the following things:

1. Retrieves an array of IDLitCommand objects from the supplied IDLitCommandSet object
2. Gets a reference to the iTool object.
3. For each command object, retrieve the identifier string for the target object. Use the identifier string to retrieve a reference to the target object itself.
4. Retrieve the OLD\_STYLE item from the command object and use its value to set the STYLE property on the target object.

### Note

---

The UndoOperation method could also have been implemented without the use of the values stored in the command set object simply by decrementing the value of the STYLE property for each target.

---

## Creating a RedoOperation Method

The operation class RedoOperation method is called when the user redoes the operation by selecting “Redo” from a menu or toolbar.

### Example RedoOperation Method

The following example code shows a very simple RedoOperation method for the ExampleOp operation:

```

FUNCTION ExampleOp::RedoOperation, oCommandSet

    ; Retrieve the IDLitCommand objects stored in the
    ; command set object.
    oCmds = oCommandSet -> Get(/ALL, COUNT = nObjs)

    ; Get a reference to the iTool object.

```

```

oTool = self -> GetTool()

; Loop through the IDLitCommand objects and restore the
; new values.
FOR i = 0, nObjs-1 DO BEGIN
    oCmds[i] -> GetProperty, TARGET_IDENTIFIER = idTarget
    oTarget = oTool -> GetByIdentifier(idTarget)
    ; Get the new value
    IF (oCmds[i] -> GetItem('NEW_STYLE', styleIndex) EQ 1) THEN $
        oTarget -> SetProperty, STYLE = styleIndex
    ENDFOR
END

```

## Discussion

The RedoOperation method does the following things:

1. Retrieves an array of IDLitCommand objects from the supplied IDLitCommandSet object
2. Gets a reference to the iTool object.
3. For each command object, retrieve the identifier string for the target object. Use the identifier string to retrieve a reference to the target object itself.
4. Retrieve the NEW\_STYLE Item from the command object and use its value to set the STYLE property on the target object.

## Note

---

The RedoOperation method could also have been implemented without the use of the values stored in the command set object simply by incrementing the value of the STYLE property for each target.

---

## Creating the Class Structure Definition

When any IDL object is created, IDL looks for an IDL class structure definition that specifies the instance data fields needed by an instance of the object, along with the data types of those fields. The object class structure must have been defined *before* any objects of the type are created. In practice, when the IDL OBJ\_NEW function attempts to create an instance of a specified object class, it executes a procedure named *ObjectClass\_\_define* (where *ObjectClass* is the name of the object), which is expected to define an IDL structure variable with the correct name and structure fields. For additional information on how IDL creates object instances, see [“The Object Lifecycle”](#) in Chapter 22 of the *Building IDL Applications* manual.

## Subclassing from the IDLitOperation Class

The IDLitOperation class is the base class for all iTool operations. In almost all cases, new operations will be subclassed either from the IDLitDataOperation class (which is itself a subclass of IDLitOperation) or from a class that is a subclass of one of these two classes.

---

### Note

If your operation acts directly on data, rather than affecting the visual appearance of objects in the iTool, you may be able to subclass from IDLitDataContainer. See [“Creating a New Data-Centric Operation”](#) on page 125 for details.

---

See [“IDLitOperation”](#) in the *IDL Reference Guide* manual for details on the methods and properties available to classes that subclass from IDLitOperation.

## Example Class Structure Definition

The following is the class structure definition for the ExampleOp operation class. This procedure should be the last procedure in a file named `exampleop__define.pro`.

```
PRO ExampleOp__Define

    struct = { ExampleOp, INHERITS IDLitOperation}

END
```

### Discussion

The purpose of the structure definition routine is to define a named IDL structure with structure fields that will contain the operation object instance data. The structure name should be the same as the operation’s class name — in this case, ExampleOp.

Like many iTool operations that act on data, ExampleOp is created as a subclass of the IDLitOperation class. The ExampleOp Operation class does not include any instance data of its own.

---

### Note

This example is intended to demonstrate how simple it can be to create a new operation class definition. While the class definition for an operation class with significant extra functionality will likely define additional structure fields, and may inherit from other iTool classes, the basic principles are the same.

---



# Registering an Operation

Before an operation can be performed by an iTool, the operation's class definition must be registered as being available to the iTool. Registering an operation with the iTool links the class definition file that contains the actual IDL code that defines the operation with a simple string that names the type. Code that performs an operation in an iTool uses the name string to specify which operation should be performed.

## Using IDLitTool::RegisterOperation

In most cases, you will register an operation with the iTool in the iTool's class Init method. Registration ensures that the operation is available to the iTool. (See [“Creating a New iTool Class”](#) on page 69 for details on the iTool class Init method.)

To register an operation, call the IDLitTool::RegisterOperation method:

```
self -> RegisterOperation, OperationName, Operation_Class_Name
```

where *OperationName* is the string you will use when referring to the operation, and *Operation\_Class\_Name* is a string that specifies the name of the class file that contains the operation's definition.

---

**Note**

The file *Operation\_Class\_Name\_\_define.pro* must exist somewhere in IDL's path for the visualization type to be successfully registered.

---

See [“IDLitTool::RegisterOperation”](#) in the *IDL Reference Guide* manual for details.

## Specifying Useful Properties

You can set any property of the [IDLitOperation](#) and [IDLitComponent](#) classes when registering an operation. The following properties may be of particular interest:

### EXPENSIVE\_OPERATION

A boolean value that indicates whether the operation is *expensive*. Expensive operations are those that require significant memory or processing time to execute. Individual operations should use the value of this property to determine whether the results of the operation should be cached to avoid re-execution when undoing or redoing.

## ICON

A string value giving the name of an icon to be associated with this object. Typically, this property is the name of a bitmap file to be used when displaying the object in a tree view. See “[Icon Bitmaps](#)” on page 28 for details on where bitmap icon files are located.

## IDENTIFIER

A string that will be used as the identifier of the object. Identifier strings specify where within an iTool’s object hierarchy an object is located; this, in turn, may affect whether and where the object is revealed in the iTool’s graphical user interface. For example, to display a menu item for an operation named 'MyOperation' in the iTool **Operations** menu, you would specify the identifier string 'operations/MyOperation'. See “[iTool Object Identifiers](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for details about how identifiers are named.

If this property is not specified, then the value of the *OperationName* argument is used as the identifier.

## REVERSIBLE\_OPERATION

A boolean value that indicates whether the operation is *reversible*. When an operation is reversible, it can be undone by applying an operation rather than restoring a stored value. Rotation by a specified angle is an example of an operation that is reversible, since applying another rotation by the same angle in the opposite direction returns the visualization to its original state. Individual operations should use the value of this property to determine how the operation should be undone.

## SHOW\_EXECUTION\_UI

A boolean value that indicates whether the operation should display a user interface element such as a dialog when the operation is executed.

## TYPES

A string or an array of strings indicating the types of data to which the operation can be applied. iTools data types are described in [Chapter 3, “Data Management”](#). Set this property to a null string ( ' ' ) to specify that the operation can be applied to all types of data.

# Unregistering an Operation

If you are creating a new `iTool` from an existing `iTool` class, you may want to remove an operation registered for the existing class from your new tool. This can be useful if you have an `iTool` class that implements all of the functionality you need, but which registers an operation you don't want included in your `iTool`. Rather than recreating the `iTool` class to remove the operation, you could create your new `iTool` class in such a way that it inherits from the existing `iTool` class, but *unregisters* the unwanted operation.

Unregister an operation by calling the `IDLitTool::UnregisterOperation` method in the `Init` method of your `iTool` class:

```
self -> UnregisterOperation, identifier
```

where *identifier* is the string value of the `IDENTIFIER` property specified when registering the operation.

For example, suppose you are creating a new `iTool` that subclasses from the standard `iSurface` tool, which is defined by the `IDLitToolSurface` class. If you wanted your new tool to behave just like the `iSurface` tool, with the exception that it would not handle the `resample` operation, you could include the following method call in your `iTool`'s `Init` method:

```
self -> UnregisterOperation, 'Operations/Transform/Resample'
```

## Finding the Identifier String

To find the string value used as the *identifier* parameter to the `UnregisterOperation` method, you must inspect the class file that registers the operation. In the case of our example, you would inspect the file `idlittoolsurface__define.pro` to find the following call to the `RegisterOperation` method:

```
self -> RegisterOperation, 'Resample', 'IDLitopResample', $  
    IDENTIFIER = 'Operations/Transform/Resample', $  
    DATA_TYPE = 'array2', ICON = 'sum'
```

The value of the `IDENTIFIER` keyword to the `RegisterOperation` method (`'Operations/Transform/Resample'`) is the string value of the operation's `IDENTIFIER` property.

# Example: Data Resample Operation

This example creates a data operation to resample data in a dataset using the IDL CONGRID function. The data resample operation is included in the file `idlitopresample__define.pro`, located in the IDL distribution in the `lib/itools/components` subdirectory of the main IDL directory.

## Class Definition File

The class definition for `idlitopresample` consists of an `Init` method, an `Execute` method, `GetProperty` and `SetProperty` methods, and a class structure definition routine. As with all object class definition files, the class structure definition routine is the last routine in the file, and the file is given the same name as the class definition routine (with the suffix `.pro` appended).

### Init Method

```
FUNCTION IDLitopResample::Init, _EXTRA = _extra

    IF (~ self -> IDLitDataOperation::Init(NAME='Resample', $
        TYPES=['IDLVECTOR', 'IDLARRAY2D', 'IDLARRAY3D'], $
        DESCRIPTION="Resampling", _EXTRA = _extra)) THEN $
        RETURN, 0

    ; Default values for resampling factors.
    self._x = 2
    self._y = 2
    self._z = 2

    ; Register properties
    self -> RegisterProperty, 'X', /FLOAT, $
        DESCRIPTION='X resampling factor.'

    self -> RegisterProperty, 'Y', /FLOAT, $
        DESCRIPTION='Y resampling factor.'

    self -> RegisterProperty, 'Z', /FLOAT, $
        DESCRIPTION='Z resampling factor.'

    self -> RegisterProperty, 'METHOD', $
        ENUMLIST=['Nearest neighbor', 'Linear', 'Cubic'], $
        NAME='Interpolation method', $
        DESCRIPTION='Interpolation method.'

    IF (N_ELEMENTS(_extra) GT 0) THEN $
```

```

self -> IDLitopResample:: SetProperty, _EXTRA = _extra

RETURN, 1

END

```

## Discussion

The first item in our class definition file is the Init method. The Init method's function signature is defined first, using the class name IDLitOpResample. The `_EXTRA` keyword inheritance mechanism allows any keywords specified in a call to the Init method to be passed through to routines that are called within the Init method even if we do not know the names of those keywords in advance.

Next, we call the Init method of the superclass. In this case, we are creating a subclass of the IDLitDataOperation class; this provides us with all of the standard iTool data operation functionality automatically. We specify three iTool data types on which our operation will work: "IDLVECTOR", "IDLARRAY2D", and "IDLARRAY3D". Any "extra" keywords specified in the call to our Init method are passed to the IDLitDataOperation::Init method via the keyword inheritance mechanism. If the call to the superclass Init method fails, we return immediately with a value of 0.

Next we store the default values for the three resampling factors (one each for the X, Y, and Z dimensions) in the object instance data fields `_x`, `_y`, and `_z`. We register each of these values as a property of the operation. We also register the METHOD property, assigning to it an enumerated list with three strings describing three different interpolation methods ("Nearest Neighbor", "Linear", and "Cubic").

If any "extra" keywords were specified in the call to our Init method, we pass them to the SetProperty method our IDLitOpResample object.

Finally, we return the value 1 to indicate successful initialization.

## Execute Method

```

FUNCTION IDLitopResample::Execute, data

  dims = SIZE(data, /DIMENSIONS)

  CASE N_ELEMENTS(dims) OF
    1: newdims = dims*ABS([self._x]) > [1]
    2: newdims = dims*ABS([self._x, self._y]) > [1, 1]
    3: newdims = dims*ABS([self._x, self._y, self._z]) > [1, 1, 1]
    ELSE: RETURN, 0

  ENDCASE

  ; No change in size.

```

```

IF (ARRAY_EQUAL(newdims, dims)) THEN RETURN, 1

interp = 0 & cubic = 0
CASE (self._method) OF
    0: ;; do nothing
    1: interp = 1
    2: cubic = 1
ENDCASE

CASE N_ELEMENTS(dims) OF
    1: data = CONGRID(data, newdims[0], $
        INTERP = interp, CUBIC = cubic)
    2: data = CONGRID(data, newdims[0], newdims[1], $
        INTERP = interp, CUBIC = cubic)
    ;; CONGRID always uses linear interp with 3D
    3: data = CONGRID(data, newdims[0], newdims[1], newdims[2])
ENDCASE

RETURN, 1

END

```

## Discussion

The `Execute` method does the work of our operation. Since `IDLitOpResample` is based on the `IDLitDataOperation` class, when the operation is requested by a user the `Execute` method is automatically called with each of the currently selected data objects as the data argument.

First, we use the `SIZE` function to determine the number of dimensions of the input data item. We use a `CASE` statement to create a new array (`newdims`) that stores the number of elements of each dimension multiplied by the scale factor for each dimension. The number of elements in each dimension cannot be less than one.

Next we use the `ARRAY_EQUAL` function to compare the number of elements of each dimension of the input data with the number of elements of each dimension of our `newdims` array. If these numbers are equal, no resampling will take place, so we stop processing and return 1 for success.

If our `newdims` array contains a different number of elements than the original input data, some resampling will take place. We check the value of the `METHOD` property (stored in the instance data field `_method`) to determine what type of resampling we should perform.

Finally, we call the `CONGRID` function with the appropriate arguments and keywords, depending on the dimensionality of the input data and the resampling method specified. We then return 1 for success.

## GetProperty Method

```

PRO IDLitopResample::GetProperty, $
  X = x, $
  Y = y, $
  Z = z, $
  METHOD = method, $
  _REF_EXTRA = _extra

  ; My properties.
  IF ARG_PRESENT(x) THEN $
    x = self._x

  IF ARG_PRESENT(y) THEN $
    y = self._y

  IF ARG_PRESENT(z) THEN $
    z = self._z

  IF ARG_PRESENT(method) THEN $
    method = self._method

  ; Superclass properties.
  IF (N_ELEMENTS(_extra) gt 0) THEN $
    self -> IDLitDataOperation::GetProperty, _EXTRA = _extra

END

```

### Discussion

The GetProperty method for our operation supports four properties named X, Y, Z, and METHOD, stored in instance data fields of the same name (with an underscore prepended). If any of these properties is specified in the call to the GetProperty method, its value is retrieved from the appropriate instance data field. Any other properties included in the method call are passed to the superclass' GetProperty method.

## SetProperty Method

```

PRO IDLitopResample::SetProperty, $
  X = x, $
  Y = y, $
  Z = z, $
  METHOD = method, $
  _EXTRA = _extra

  ; My properties.
  IF N_ELEMENTS(x) THEN $

```

```

        IF (x NE 0) THEN self._x = x

    IF N_ELEMENTS(y) THEN $
        IF (y NE 0) THEN self._y = y

    IF N_ELEMENTS(z) THEN $
        IF (z NE 0) THEN self._z = z

    IF N_ELEMENTS(method) THEN $
        self._method = method

; Superclass properties.
IF (N_ELEMENTS(_extra) gt 0) THEN $
    self -> IDLitDataOperation::SetProperty, _EXTRA = _extra

END

```

## Discussion

The SetProperty method for our operation supports four properties named X, Y, Z, and METHOD, stored in instance data fields of the same name (with an underscore prepended). If any of these properties is specified in the call to the SetProperty method, its value is stored in the appropriate instance data field. Any other properties included in the method call are passed to the superclass' SetProperty method.

## Class Definition

```

PRO IDLitopResample__define

    struc = {IDLitopResample, $
        inherits IDLitDataOperation,    $
        _x: 0d, $
        _y: 0d, $
        _z: 0d, $
        _method: 0b $
    }

END

```

## Discussion

Our class definition routine is very simple. We create an IDL structure variable with the name IDLitOpResample, specifying that the structure inherits from the IDLitDataOperation class. The structure has three instance data fields named \_x, \_y, and \_z, which contain double-precision floating point values, and a single instance data field named \_method which contains a byte value.





# Chapter 8: Creating a File Reader

This chapter describes the process of creating an iTool file reader.

---

Overview .....	162	Registering a File Reader .....	177
Predefined iTool File Readers .....	163	Unregistering a File Reader .....	178
Creating a New File Reader .....	166	Example: TIFF File Reader .....	179

# Overview

A *file reader* is an iTool component object class that defines how data stored in a file should be imported into the iTool environment. File readers have mechanisms for determining the type of data stored in a file, which allows them to create IDLitData objects from the stored data. Some file readers implement a graphical user interface allowing the user to specify the format of data before importing into the iTool; others read a well-defined file type and operate more or less automatically. Some examples of iTool file readers are:

- the ASCII file reader, which uses the IDL ASCII\_TEMPLATE and READ\_ASCII functions to allow the user to define the format of data in a text file,
- various image file readers, which allow the user to import data stored in JPEG, BMP, PNG, and other well-defined image format files,
- a generic binary file reader, which allows the user to specify the format of files containing binary data.

A number of standard file readers are predefined and included in the IDL iTools package; if none of the predefined file readers suits your needs, you can create your own file reader by subclassing either from the base IDLitReader class on which all of the predefined file readers are based, or from one of the predefined file readers.

## The File Reader Creation Process

To create a new iTool file reader, you will do the following:

- Choose an iTool file reader class on which your new operation will be based. In almost all cases, you will base your new operation on the IDLitReader class, which handles registration of standard file properties and provides standard messaging features.
- Provide methods to check the type of data stored in the file and place the retrieved the data in a data object.
- Set data object properties.

This chapter describes the process of creating a new file reader based on the IDLitReader class.

# Predefined iTool File Readers

The iTool system distributed with IDL includes a number of pre-defined file readers. You can include these file readers in an iTool directly by registering the class with your iTool (as described in “[Registering a File Reader](#)” on page 177). You can also create a new file reader class based on one of the pre-defined classes.

## IDLitReadASCII

The iTools ASCII file reader uses the IDL `READ_ASCII` and `ASCII_TEMPLATE` functions to read data from an ASCII file into an IDL variable or variables. It presents a *wizard* interface that allows the user to define the structure of the data in the ASCII file and specify which data should be included.

### Registered Properties

None

## IDLitReadBinary

The iTools Binary file reader uses the IDL `READ_BINARY` and `BINARY_TEMPLATE` functions to read data from a binary data file into an IDL variable or variables. It presents a *wizard* interface that allows the user to define the structure of the data in the binary file and specify which data should be included.

### Registered Properties

**TEMPLATE** — A template structure (previously defined by the `BINARY_TEMPLATE` function) describing the file to be read.

## IDLitReadBMP

The iTools BMP file reader uses the IDL `READ_BMP` function to read a `*.bmp` file and place the image data in an iTool image data object.

### Registered Properties

None

## IDLitReadDICOM

The iTools DICOM reader uses the IDL `READ_DICOM` function to read a `*.dcm` and place the image data in an iTool image data object.

### Registered Properties

None

## IDLitReadISV

The iTools Saved Variables file reader restores a saved iTool state (`*.isv`) file. All data objects in the file are placed into the current iTool data manager session, and all visualization objects are restored and displayed.

### Registered Properties

None

## IDLitReadJPEG

The iTools JPEG file reader uses the IDL `READ_JPEG` procedure to read a `*.jpg` or `*.jpeg` file and place the image data in an iTool image data object.

### Registered Properties

None

## IDLitReadPICT

The iTools PICT file reader uses the IDL `READ_PICT` procedure to read a `*.pct` or `*.pict` file and place the image data in an iTool image data object.

### Registered Properties

None

## IDLitReadPNG

The iTools PNG file reader uses the IDL `READ_PNG` function to read a `*.png` file and place the image (and, optionally, palette) data in an iTool image data object.

### Registered Properties

None

## IDLitReadTIFF

The iTools TIFF file reader uses the IDL `READ_TIF` function to read a `*.tif` or `*.tiff` file and place the image (and, optionally, palette) data in an iTool image data object.

### Registered Properties

**IMAGE\_INDEX** — An integer specifying the index of the image within the TIFF file that should be read into the image data object.

### IDLitReadWAV

The iTools WAV file reader uses the IDL `READ_WAV` function to read a `*.wav` file and place the data in an iTool vector object.

### Registered Properties

None

# Creating a New File Reader

An iTool file reader class definition file must (at the least) provide methods to initialize the file reader class, get and set property values, handle changes to the underlying data, clean up when the file reader is destroyed, and define the file reader class structure. Complex file reader types will likely provide additional methods.

The process of creating an file reader is outlined in the following sections:

- [“Creating an Init Method”](#) on page 166
- [“Creating a Cleanup Method”](#) on page 170
- [“Creating a GetProperty Method”](#) on page 171
- [“Creating a SetProperty Method”](#) on page 172
- [“Creating an IsA Method”](#) on page 173
- [“Creating a GetData Method”](#) on page 174
- [“Creating the Class Structure Definition”](#) on page 175

## Creating an Init Method

The file reader class Init method handles any initialization required by the file reader object, and should do the following:

- define the Init function method
- call the Init methods of any superclasses
- register any properties of your file reader, and set property attributes as necessary
- perform other initialization steps as necessary
- return the value 1 if the initialization steps are successful, or 0 otherwise

### Definition of the Init Function

Begin by defining the argument and keyword list for your Init method. The argument and keyword list defines positional parameters (arguments) accepted by your method, defines any keywords that will be handled directly by your method, and specifies whether keywords not explicitly handled by your method will be passed through to other routines called by your method via IDL’s keyword inheritance mechanism. The function signature for an Init method for a file reader generally looks something like this:

```
FUNCTION MyReader::Init, MYKEYWORD1 = mykeyword1, $
    MYKEYWORD2 = mykeyword2, ..., _REF_EXTRA = _extra
```

where *MyReader* is the name of your file reader class and the *MYKEYWORD* parameters are keywords handled explicitly by your Init function.

Use keyword inheritance (the `_REF_EXTRA` keyword) to pass keyword parameters through to any called routines as necessary. (See “[Keyword Inheritance](#)” in Chapter 4 of the *Building IDL Applications* manual for details on IDL’s keyword inheritance mechanism.)

## Superclass Initialization

The file reader class Init method should call the Init method of any required superclass. For example, if your file reader is based on an existing file reader class, you would call that class’ Init method:

```
success = self -> SomeFileReaderClass::Init(_EXTRA = _extra)
```

where *SomeFileReaderClass* is the class definition file for the file reader on which your new file reader is based. The variable `success` will contain a 1 if the initialization was successful.

### Note

---

Your file reader class may have multiple superclasses. In general, each superclass’ Init method should be invoked by your class’ Init method.

---

## Error Checking

Rather than simply calling the superclass Init method, it is a good idea to check whether the call to the superclass Init method succeeded. The following statement checks the value returned by the superclass Init method; if the returned value is 0 (indicating failure), the current Init method also immediately returns with a value of 0:

```
IF (self -> SomeFileReaderClass::Init() EQ 0) THEN RETURN, 0
```

This convention is used in all file reader classes included with IDL. RSI strongly suggests that you include similar checks in your own class definition files.

## Keywords to the Init Method

Properties of the file reader class can be set in the Init method by specifying the property names and values as IDL keyword-value pairs. In addition to any keywords implemented directly in the Init method of the superclass on which you base your class, the properties of the `IDLitReader` class and the `IDLitComponent` class are

available to any file reader class. See [“IDLitReader Properties”](#) and [“IDLitComponent Properties”](#) in the *IDL Reference Guide* manual.

Use keyword inheritance (the `_EXTRA` keyword) to pass keyword parameters through to the superclass as necessary. (See [“Keyword Inheritance”](#) in Chapter 4 of the *Building IDL Applications* manual for details on IDL’s keyword inheritance mechanism.)

## Standard Base Class

While you can create your new file reader class from any existing file reader class, in many cases, file reader classes you create will be subclassed directly from the base class `IDLitReader`:

```
IF (self -> IDLitReader::Init(Extensions, _EXTRA = _extra) EQ 0) $
    THEN RETURN, 0
```

where *Extensions* is a string or array of strings specifying the filename extensions readable by your file reader.

### Note

---

The value of the *Extensions* argument is used only to display the proper filename filter when an Open dialog is displayed — it is not a check for the proper filetype. The `IsA` method must check the file to determine whether it is readable by your file reader.

---

The `IDLitReader` class provides the base `iTool` file reader functionality used in the tools created by RSI. See [“Subclassing from the IDLitReader Class”](#) on page 175 for details.

## Return Value

If all of the routines and methods used in the `Init` method execute successfully, it should indicate successful initialization by returning 1. Other file reader classes that subclass from your file reader class may check this return value, as your routine should check the value returned by any superclass `Init` methods called.

## Registering Properties

File reader objects can register properties with the `iTool`; registered properties show up in the property sheet interface shown in the *system preferences browser* (described in [“Properties of the iTools System”](#) on page 64), and can be modified interactively by users. The `iTool` property interface is described in detail in [Chapter 4, “Property Management”](#).



Register a property by calling the RegisterProperty method of the IDLitComponent class:

```
self -> RegisterProperty, PropertyIdentifier [, TypeCode] $
    [, ATTRIBUTE = value]
```

where *PropertyIdentifier* is a string that uniquely identifies the property, *TypeCode* is an integer between 0 and 9 specifying the property data type, and *ATTRIBUTE* is a property attribute. See “[Registering Properties](#)” on page 54 for details.

---

### Note

A file reader need not register any properties at all, if the read operation is simple. Many of the standard iTool image file readers work without registering any properties.

---

## Setting Property Attributes

If a property has already been registered, perhaps by a superclass of your file reader class, you can change the registered attribute values using the SetPropertyAttribute method of the IDLitComponent class:

```
self -> SetPropertyAttribute, Identifier
```

where *Identifier* is the name of the keyword to the GetProperty and SetProperty methods used to retrieve or change the value of this property. The Identifier is specified in the call to RegisterProperty either via the *PropertyName* argument or the IDENTIFIER keyword. See “[Property Attributes](#)” on page 58 for additional details.

## Passing Through Caller-Supplied Property Settings

If you have included the `_REF_EXTRA` keyword in your function definition, you can use IDL’s keyword inheritance mechanism to pass any “extra” keyword values included in the call to the Init method through to other routines. This mechanism allows you to specify property settings when the Init method is called; simply include each property’s keyword/value pair when calling the Init method, and include the following in the body of the Init method:

```
IF (N_ELEMENTS(_extra) GT 0) THEN $
    self -> MyReader::SetProperty, _EXTRA = _extra
```

where *MyReader* is the name of your file reader class. This line has the effect of passing any “extra” keyword values to your file reader class’ SetProperty method, where they can either be handled directly or passed through to the SetProperty methods of the superclasses of your class. See “[Creating a SetProperty Method](#)” on page 172 for details.

## Example Init Method

```
FUNCTION ExampleReader::Init, _EXTRA = _extra

    IF (self -> IDLiteReader::Init('ppm', FILETYPE='PPM', $
        DESCRIPTION="PPM File Reader", $
        _EXTRA = _extra) EQ 0) THEN $
        RETURN, 0

    RETURN, 1

END
```

### Discussion

The `ExampleReader` class is based on the `IDLiteReader` class (discussed in “[Subclassing from the IDLiteReader Class](#)” on page 175). As a result, all of the standard features of an `iTool` file reader class are already present. We don’t define any keyword values to be handled explicitly in the `Init` method, but we do use the keyword inheritance mechanism to pass keyword values through to methods called within the `Init` method. The `ExampleReader` `Init` method does the following things:

1. Calls the `Init` method of the superclass, `IDLiteReader`. We specify a list of accepted filename extensions (only `ppm`, in this case) via the *Extensions* argument, and set the `FILETYPE` keyword. We include a description of the reader via the `DESCRIPTION` keyword. Finally, we use the `_EXTRA` keyword inheritance mechanism to pass through any keywords provided when the `ExampleReader` `Init` method is called.
2. Returns the integer 1, indicating successful initialization.

## Creating a Cleanup Method

The file reader class `Cleanup` method handles any cleanup required by the file reader object, and should do the following:

- destroy any pointers or objects created by the file reader
- call the superclass’ `Cleanup` method

Calling the superclass’ `cleanup` method will destroy any objects created when the superclass was initialized.

### Note

If your file reader class is based on the `IDLiteReader` class, and does not create any pointers or objects of its own, the `Cleanup` method is not strictly required. It is

always safest, however, to create a Cleanup method that calls the superclass' Cleanup method.

---

See “[IDLitReader::Cleanup](#)” in the *IDL Reference Guide* manual for additional details.

## Example Cleanup Method

```
PRO ExampleReader::Cleanup

    ;; Cleanup superclass
    self -> IDLitReader::Cleanup

END
```

### Discussion

Since our file reader object does not have any instance data of its own, the Cleanup method simply calls the superclass Cleanup method.

## Creating a GetProperty Method

The file reader class GetProperty method retrieves property values from the file reader object instance or from instance data of other associated objects. It should retrieve the requested property value, either from the file reader object's instance data or by calling another class' GetProperty method.

### Note

---

Any property registered with a call to the RegisterProperty method must be listed as a keyword to the GetProperty method either of the visualization class or one of its superclasses.

---

### Note

---

A file reader need not register any properties at all, if the read operation is simple. Many of the standard iTool image file readers work without registering any properties.

---

See “[IDLitReader::GetProperty](#)” in the *IDL Reference Guide* manual for additional details.

## Example GetProperty Method

```

PRO ExampleReader::GetProperty, _REF_EXTRA = _extra

    IF (N_ELEMENTS(_extra) GT 0) THEN $
        self -> IDLitReader::GetProperty, _EXTRA = _extra

END

```

### Discussion

The `GetProperty` method first defines the keywords it will accept. There must be a keyword for each property of the file reader. Since the file reader we are creating has no properties of its own, there are no keywords explicitly defined. The keyword inheritance mechanism allows properties to be retrieved from the `ExampleReader` class' superclasses without knowing the names of the properties.

Since our `ExampleReader` class has no properties of its own, we simply call the superclass' `GetProperty` method, passing in all of the keywords stored in the `_EXTRA` structure.

## Creating a SetProperty Method

The file reader `SetProperty` method stores property values in the file reader object's instance data. It should set the specified property value, either by storing the value directly in the visualization object's instance data or by calling another class' `SetProperty` method.

### Note

---

Any property registered with a call to the `RegisterProperty` method must be listed as a keyword to the `SetProperty` method either of the visualization class or one of its superclasses.

---

### Note

---

A file reader need not register any properties at all, if the read operation is simple. Many of the standard iTool image file readers work without registering any properties.

---

See [“IDLitReader::SetProperty”](#) in the *IDL Reference Guide* manual for additional details.

## Example SetProperty Method

```
PRO ExampleReader::SetProperty, _EXTRA = _extra

    IF (N_ELEMENTS(_extra) GT 0) THEN $
        self -> IDLitReader::SetProperty, _EXTRA = _extra

END
```

### Discussion

The SetProperty method first defines the keywords it will accept. There must be a keyword for each property of the visualization type. Since the file reader we are creating has no properties of its own, no keywords are explicitly defined. The keyword inheritance mechanism allows properties to be set on the ExampleReader class' superclasses without knowing the names of the properties.

Using the N\_ELEMENTS function, we check to see whether any properties were specified via the keyword inheritance mechanism. If any keywords were specified, we call the superclass' SetProperty method, passing in all of the keywords stored in the \_EXTRA structure.

## Creating an IsA Method

The file reader IsA method must accept a string containing the name of the file to be read as its only parameter, and must determine whether the file is of the proper type to be read by your file reader. If the file is of the correct type, the IsA method must return 1; if the file is not of the correct type, the IsA method should display an error message and return 0.

See “IDLitReader::IsA” in the *IDL Reference Guide* manual for additional details.

### Example IsA Method

```
FUNCTION ExampleReader::IsA, strFilename

    iDot = STRPOS(strFilename, '.', /REVERSE_SEARCH)

    IF (iDot GT 0) THEN BEGIN
        fileSuffix = STRUPCASE(STRMID(strFilename, iDot + 1))
        IF (STRUPCASE(fileSuffix) EQ 'PPM') THEN RETURN, 1
    ENDIF

    self -> IDLitIMessaging::ErrorMessage, $
        ["The specified file is not a PPM file."], $
        SEVERITY = 0, TITLE="Wrong File Type"
```

```
RETURN, 0
```

```
END
```

## Discussion

### Note

---

Our example IsA method will simply check the filename for the presence of the proper filename extension. A more sophisticated IsA method would actually inspect the contents of the specified file.

---

The IsA method accepts a string that contains a file name. Using the supplied file name, we first search backwards from the end of the name until we locate a dot character. If the filename contains a dot, we extract the string that follows the dot and convert it to upper case. If the extracted string is 'PPM', we return success; if the extracted string is not 'PPM' or if there is no dot in the file name, we issue an error using the `IDLitMessaging::ErrorMessage` method and return failure.

## Creating a GetData Method

The file reader `GetData` method does the work of the file reader, first creating an IDL variable or variables to contain the data read from the file, then placing the data into an iTool data object. If this process is successful, the `GetData` method must place the created data object in the variable supplied as the method's only argument and return 1 for success. If the process is not successful, the `GetData` method must return 0.

See “[IDLitReader::GetData](#)” in the *IDL Reference Guide* manual for additional details.

### Example GetData Method

```
FUNCTION ExampleReader::GetData, oImageData

; Get the name of the file currently associated with the reader.
filename = self -> GetFilename()

; Read the file.
READ_PPM, filename, image

; Store image data in Image Data object.
oImageData = OBJ_NEW('IDLitDataIDLImage', $
    NAME = FILE_BASENAME(fileName))

IF OBJ_VALID(oImageData) THEN BEGIN
    RETURN, oImageData -> SetData(image, 'ImagePixels', /NO_COPY)
ENDIF
```

```
RETURN, 0
```

```
END
```

## Discussion

The `GetData` method accepts a single argument, which is a named variable that will contain the data object. Our `GetData` method's first step is to retrieve the file name of the file on which the method is being called using the `GetFilename` method. Since our example file reader reads data from PPM files, the file name is then passed to the IDL `READ_PPM` procedure. An `IDLitDataImage` object that will hold the image data is created in the named variable specified by the argument to the `GetData` method (`oImageData`, in this case); the `NAME` property set to the filename of the original data file. We check to ensure that the `oImageData` object was created successfully and add the image data returned by the `READ_PPM` procedure using the `IDLitData::SetData` method. Note the use of the `NO_COPY` keyword to prevent making copies of the image data array, which could be quite large. Finally, we return the value returned by the `SetData` method (1 for success, 0 for failure), or we return 0 if `oImageData` is not a valid object.

## Creating the Class Structure Definition

When any IDL object is created, IDL looks for an IDL class structure definition that specifies the instance data fields needed by an instance of the object, along with the data types of those fields. The object class structure must have been defined *before* any objects of the type are created. In practice, when the IDL `OBJ_NEW` function attempts to create an instance of a specified object class, it executes a procedure named `ObjectClass__define` (where *ObjectClass* is the name of the object), which is expected to define an IDL structure variable with the correct name and structure fields. For additional information on how IDL creates object instances, see [“The Object Lifecycle”](#) in Chapter 22 of the *Building IDL Applications* manual.

## Subclassing from the IDLitReader Class

The `IDLitReader` class is the base class for all *iTool* file readers. In almost all cases, new file readers will be subclassed either from the `IDLitReader` class or from a class that is a subclass of `IDLitReader`.

See [“IDLitReader”](#) in the *IDL Reference Guide* manual for details on the methods and properties available to classes that subclass from `IDLitReader`.

## Example Class Structure Definition

The following is the class structure definition for the `ExampleReader` file reader class. This procedure should be the last procedure in a file named `examplereader__define.pro`.

```
PRO ExampleReader__Define

    struct = { ExampleReader,          $
              INHERITS IDLitReader $
            }

END
```

### Discussion

The purpose of the structure definition routine is to define a named IDL structure with structure fields that will contain the visualization object instance data. The structure name should be the same as the visualization's class name — in this case, `ExampleReader`.

Like many `iTool` file reader classes, `ExampleReader` is created as a subclass of the `IDLitReader` class. File reader classes that subclass from `IDLitReader` class inherit all of the standard `iTool` file reader features, as described in [“Subclassing from the IDLitReader Class”](#) on page 175.

The `ExampleReader` class has no instance data of its own. For a more complex example, see [“Example: TIFF File Reader”](#) on page 179.



# Registering a File Reader

Before a file reader can be used by an iTool to read in a file, the file reader's class definition must be registered as being available to the iTool. Registering a file reader with the iTool links the class definition file that contains the actual IDL code that defines the file reader with a simple string that names the reader. Code that calls a file reader in an iTool uses the name string to specify which reader should be created.

## Using IDLitTool::RegisterFileReader

In most cases, you will register a file reader with the iTool in the iTool's class Init method. Registration ensures that the file reader is available when the iTool attempts to use it to read a file. (See “Creating a New iTool Class” on page 69 for details on the iTool class Init method.)

To register a file reader, call the IDLitTool::RegisterFileReader method:

```
self -> RegisterFileReader, Reader_Type, ReaderType_Class_Name, $  
      ICON = icon
```

where *Reader\_Type* is the string you will use when referring to the file reader, *ReaderType\_Class\_Name* is a string that specifies the name of the class file that contains the file reader's definition, and *icon* is a string containing the name of a bitmap file to be used in the preferences browser.

### Note

---

The file *ReaderType\_Class\_Name\_\_define.pro* must exist somewhere in IDL's path for the file reader to be successfully registered.

---

See “IDLitTool::RegisterFileReader” in the *IDL Reference Guide* manual for details.

## Specifying Useful Properties

You can set any property of the [IDLitReader](#) and [IDLitComponent](#) classes when registering a file reader. The following properties may be of particular interest:

### ICON

A string value giving the name of an icon to be associated with this object. Typically, this property is the name of a bitmap file to be used when displaying the object in a tree view. See “[Icon Bitmaps](#)” on page 28 for details on where bitmap icon files are located.

# Unregistering a File Reader

If you are creating a new `iTool` from an existing `iTool` class, you may want to remove a file reader registered for the existing class from your new tool. This can be useful if you have an `iTool` class that implements all of the functionality you need, but which registers a file reader you don't want included in your `iTool`. Rather than recreating the `iTool` class to remove the file reader, you could create your new `iTool` class in such a way that it inherits from the existing `iTool` class, but *unregisters* the unwanted file reader.

Unregister a file reader by calling the `IDLitTool::UnregisterFileReader` method in the `Init` method of your `iTool` class:

```
self -> UnregisterFileReader, identifier
```

where *identifier* is the string name used when registering the file reader.

For example, suppose you are creating a new `iTool` that subclasses from a standard `iTool` that is based on the `IDLitToolbase` class. If you wanted your new tool to behave just like the a standard tool, with the exception that it would not read PNG files, you could include the following method call in your `iTool`'s `Init` method:

```
self -> UnregisterFileReader, 'PNG File Reader'
```

## Finding the Identifier String

To find the string value used as the *identifier* parameter to the `UnregisterFileReader` method, you must inspect the class file that registers the file reader. In the case of our example, you would inspect the file `idlittoolbase__define.pro` to find the following call to the `RegisterFileReader` method:

```
self -> RegisterFileReader, 'PNG File Reader', 'IDLitReadPNG'
```

The first argument to the `RegisterFileReader` method (`'PNG File Reader'`) is the string name of the file reader.

## Example: TIFF File Reader

This example creates a file reader to read TIFF format files. The TIFF file reader is included in the file `idlitreadtiff__define.pro`, located in the IDL distribution in the `lib/itools/components` subdirectory of the main IDL directory.

### Class Definition File

The class definition for `idlitreadtiff` consists of an `Init` method, an `IsA` method, a `GetData` method, `GetProperty` and `SetProperty` methods, and a class structure definition routine. As with all object class definition files, the class structure definition routine is the last routine in the file, and the file is given the same name as the class definition routine (with the suffix `.pro` appended).

#### Init Method

```

FUNCTION IDLitReadTIFF::Init, _EXTRA = _extra

    ; Call the superclass Init method
    IF (self -> IDLitReader::Init(["tiff", "tif"], $
        FILETYPE="TIFF", NAME="Tiff Files", $
        DESCRIPTION="TIFF File format", $
        _EXTRA = _extra) NE 1) THEN $
    RETURN, 0

    ; Initialize the instance data field
    self._index = 0

    ; Register the index property
    self -> RegisterProperty, 'IMAGE_INDEX', /INTEGER, $
        Description='Index of the image to read from the TIFF file.'

    RETURN, 1

END

```

#### Discussion

The first item in our class definition file is the `Init` method. The `Init` method's function signature is defined first, using the class name `IDLitReadTIFF`. The `_EXTRA` keyword inheritance mechanism allows any keywords specified in a call to the `Init` method to be passed through to routines that are called within the `Init` method even if we do not know the names of those keywords in advance.

Next, we call the `Init` method of the superclass. In this case, we are creating a subclass of the `IDLitReader` class; this provides us with all of the standard `iTool` file reader functionality automatically. Any “extra” keywords specified in the call to our `Init` method are passed to the `IDLitReader::Init` method via the keyword inheritance mechanism.

We specify a list of accepted filename extensions (`tiff` and `tif`, in this case) via the `Extensions` argument, and set the `FILETYPE` keyword. We specify a value for the `NAME` property of the reader object (this is displayed in the system preferences dialog) and include a description of the reader via the `DESCRIPTION` keyword. Finally, we use the `_EXTRA` keyword inheritance mechanism to pass through any keywords provided when the `Init` method is called.

Our TIFF reader object has a single instance data field: `_index`, which is used to store the index number of the image to read from a multi-image TIFF file. We initialize this instance data field to 0, and register the `IMAGE_INDEX` property to provide access to this field via the property sheet interface.

Finally, we return the value 1 to indicate successful initialization.

## IsA Method

```
FUNCTION IDLitReadTIFF::Isa, strFilename
    RETURN, QUERY_TIFF(strFilename);
END
```

## Discussion

The `IsA` method for our TIFF file reader is simple: we use the IDL `QUERY_TIFF` function to determine whether the specified file is a TIFF file, returning the function’s return value.

## GetData Method

```
FUNCTION IDLitReadTIFF::GetData, oImageData
    filename = self -> GetFilename()
    IF (QUERY_TIFF(filename, fInfo, IMAGE_INDEX = self._index) EQ 0)
    $
        THEN RETURN, 0
    IF (fInfo.has_palette) THEN BEGIN
        image = READ_TIFF(filename, palRed, palGreen, palBlue, $
            IMAGE_INDEX = self._index)
```

```

ELSE
    image = READ_TIFF(filename, IMAGE_INDEX = self._index)
ENDIF

; Store image data in Image Data object.
oImageData = OBJ_NEW('IDLitDataImage', $
    NAME = FILE_BASENAME(fileName))

result = oImageData -> SetData(image, 'Image', /NO_COPY)

IF (RESULT EQ 0) THEN $
    RETURN, 0

; Store palette data in Image Data object.
IF (fInfo.has_palette) THEN $
    result = oImageData -> SetData( TRANSPOSE([[palRed], $
        [palGreen], [palBlue]]), 'Palette')

IF fInfo.num_images GT 1 THEN $
    self -> IDLitIMessaging::StatusMessage, $
        'Read channel '+strtrim(self._index,2)

RETURN, result

END

```

## Discussion

The `GetData` method for our TIFF file reader begins by retrieving the name of the file associated with the reader object. We then use the `IDL QUERY_TIFF` function to check whether the image specified by the value of the `IMAGE_INDEX` property (stored in the `_index` instance data field) exists, returning 0 for failure if the specified image does not exist.

`QUERY_TIFF` also returns a structure containing information about the image; we use this structure to determine whether the image has a palette. We use the presence of a palette to choose the correct call to the `READ_TIFF` function, which places the image data in a set of local variables.

Next, we construct an `IDLitDataImage` object to store the image data, using the base name of the image file for the object's `NAME` property. We use the `SetData` method to place the image data into the newly created image data object, specifying the string `'Image'` as the data object's identifier. A check of the return value from the `SetData` method allows us to return 0 from our `GetData` method if we are unable to store the image data in the image object for any reason.

If the image includes palette data, we store the array of red, green, and blue values using the `SetData` method, specifying `'Palette'` as the identifier. The palette

variables returned by `READ_TIFF` represent image *planes*; since the `IDLitVisImage` visualization type that we will use to display the image expects data interleaved by pixel, we use the `TRANSPPOSE` function to convert the palette data into the correct format.

Finally, we use the `StatusMessage` method of the `IDLitIMessaging` class to report to the user which image was retrieved from the TIFF file. The message is displayed in the status area of the `iTool` window.

## GetProperty Method

```
PRO IDLitReadTIFF::GetProperty, IMAGE_INDEX = IMAGE_INDEX, $
    _REF_EXTRA = _extra

    IF (ARG_PRESENT(image_index)) THEN $
        image_index= self._index

    IF (N_ELEMENTS(_extra) GT 0) THEN $
        self -> IDLitReader::GetProperty, _extra = _extra

END
```

### Discussion

The `GetProperty` method for our TIFF file reader supports a single property named `IMAGE_INDEX`. If this property is specified in the call to the `GetProperty` method, its value is retrieved from the `_index` instance data field. Any other properties included in the method call are passed to the superclass' `GetProperty` method.

## SetProperty Method

```
PRO IDLitReadTIFF::SetProperty, IMAGE_INDEX = IMAGE_INDEX, $
    _EXTRA = _extra

    IF (N_ELEMENTS(image_index) GT 0) THEN $
        self._index = image_index

    IF (N_ELEMENTS(_extra) GT 0) THEN $
        self -> IDLitReader::SetProperty, _extra = _extra

END
```

### Discussion

The `SetProperty` method for our TIFF file reader supports a single property named `IMAGE_INDEX`. If this property is specified in the call to the `SetProperty` method, its value is placed in the `_index` instance data field. Any other properties included in the method call are passed to the superclass' `SetProperty` method.

## Class Definition

```
PRO IDLitReadTIFF__Define

    struct = {IDLitReadTIFF,          $
              inherits IDLitReader, $
              _index : 0             $
            }

END
```

### Discussion

Our class definition routine is very simple. We create an IDL structure variable with the name `IDLitReadTIFF`, specifying that the structure inherits from the `IDLitReader` class. The structure has a single instance data field named `_index`, which we specify as an integer value.







# Chapter 9: Creating a File Writer

This chapter describes the process of creating an iTool file writer.

---

Overview .....	186	Unregistering a File Writer .....	202
Creating a New File Writer .....	190	Example: TIFF File Writer .....	203
Registering a File Writer .....	201		

# Overview

A *file writer* is an iTool component object class that defines how data stored in the iTool data manager can be exported to a file. File writers have mechanisms for manipulating data stored in iTool data objects into the proper format for a given file type. Some examples of iTool file writers are:

- the ASCII file writer, which uses the IDL PRINTF procedure to write data to a text file.
- various image file writers, which allow the user to save data in JPEG, BMP, PNG, and other well-defined image format files,
- a generic binary file writer, which uses the IDL WRITEU procedure to write unformatted binary data to a file.

A number of standard file writers are predefined and included in the IDL iTools package; if none of the predefined file writers suits your needs, you can create your own file writer by subclassing either from the base IDLitWriter class on which all of the predefined file writers are based, or from one of the predefined file writers.

## The File Writer Creation Process

To create a new iTool file writer, you will do the following:

- Choose an iTool file writer class on which your new operation will be based. In almost all cases, you will base your new operation on the IDLitWriter class, which handles registration of standard file properties and provides standard messaging features.
- Provide methods that extract the image data from the data object and create a file using IDL's output routines (PRINT, WRITE, or one of the IDL WRITE\_\* routines).

This chapter describes the process of creating a new file writer based on the IDLitWriter class.

# Predefined iTool File Writers

The iTool system distributed with IDL includes a number of pre-defined file writers. You can include these file writers in an iTool directly by registering the class with your iTool (as described in “[Registering a File Writer](#)” on page 201). You can also create a new file writer class based on one of the pre-defined classes.

## IDLitWriteASCII

The iTools ASCII file writer uses the IDL PRINTF procedure to print strings to a file.

### Registered Properties

**STRING\_SEPARATOR** — A string that is used to separate the values stored in the ASCII file.

**USE\_DEFAULT\_FORMAT** — A boolean value that specifies whether a default format string should be used.

**STRING\_FORMAT** — A string specifying the format string to be used when writing the data to the ASCII file. See “[Format Codes](#)” in [Chapter 10](#), “[Files and Input/Output](#)” in the *Building IDL Applications* manual for a discussion of format codes.

### Note

---

The format code should not include parentheses.

---

## IDLitWriteBinary

The iTools Binary file writer uses the IDL WRITEU procedure to write unformatted binary data to a file.

### Registered Properties

None

## IDLitWriteBMP

The iTools BMP file writer uses the IDL WRITE\_BMP procedure to write an image and its color table vectors to a Microsoft Windows Version 3 device independent bitmap file (.bmp).

### Registered Properties

None

## IDLitWriteISV

The iTools ISV file writer saves the current iTool state, including data in the data manager, visualizations, annotations, and operation property settings to a file with the extension `.isv`. ISV files can be restored by launching an iTool and selecting the file using the **File** → **Open** menu item.

### Registered Properties

None

## IDLitWriteJPEG

The iTools JPEG file writer uses the IDL `WRITE_JPEG` procedure to write compressed images to files. JPEG (Joint Photographic Experts Group) is a standardized compression method for full-color and gray-scale images.

### Registered Properties

**QUALITY** — 1An integer specifying the quality index, in the range of 0 (terrible) to 100 (excellent) for the JPEG file. The default value is 75, which corresponds to very good quality. Lower values of **QUALITY** produce higher compression ratios and smaller files.

## IDLitWritePICT

The iTools PICT file writer uses the IDL `WRITE_PICT` procedure to write an image and its color table vectors to a PICT (version 2) format image file. The PICT format is used by Apple Macintosh computers.

### Registered Properties

None

## IDLitWritePNG

The iTools PNG file writer uses the IDL `WRITE_PNG` procedure to write an image to a Portable Network Graphics (PNG) file. The data in the file is stored using lossless compression with either 8 or 16 data bits per channel, based on the input IDL variable type.

### Registered Properties

None

## **IDLitWriteTIFF**

The iTools TIFF file writer uses the IDL WRITE\_TIFF procedure to write TIFF files.

### **Registered Properties**

None

# Creating a New File Writer

The process of creating an visualization type is outlined in the following sections:

- “[Creating an Init Method](#)” on page 190
- “[Creating a Cleanup Method](#)” on page 194
- “[Creating a GetProperty Method](#)” on page 195
- “[Creating a SetProperty Method](#)” on page 196
- “[Creating a SetData Method](#)” on page 197
- “[Creating the Class Structure Definition](#)” on page 199

## Creating an Init Method

The file writer class Init method handles any initialization required by the file writer object, and should do the following:

- define the Init function method
- call the Init methods of any superclasses
- register any properties of your file writer, and set property attributes as necessary
- perform other initialization steps as necessary
- return the value 1 if the initialization steps are successful, or 0 otherwise

### Definition of the Init Function

Begin by defining the argument and keyword list for your Init method. The argument and keyword list defines positional parameters (arguments) accepted by your method, defines any keywords that will be handled directly by your method, and specifies whether keywords not explicitly handled by your method will be passed through to other routines called by your method via IDL’s keyword inheritance mechanism. The Init method for a file writer generally looks something like this:

```
FUNCTION MyWriter::Init, MYKEYWORD1 = mykeyword1, $  
    MYKEYWORD2 = mykeyword2, ..., _REF_EXTRA = _extra
```

where *MyWriter* is the name of your file writer class and the *MYKEYWORD* parameters are keywords handled explicitly by your Init function.

Use keyword inheritance (the `_REF_EXTRA` keyword) to pass keyword parameters through to any called routines as necessary. (See “[Keyword Inheritance](#)” in Chapter 4

of the *Building IDL Applications* manual for details on IDL's keyword inheritance mechanism.)

## Superclass Initialization

The file writer class Init method should call the Init method of any required superclass. For example, if your file writer is based on an existing file writer class, you would call that class' Init method:

```
self -> SomeFileWriterClass::Init(_EXTRA = _extra)
```

where *SomeFileWriterClass* is the class definition file for the file writer on which your new file writer is based.

### Note

---

Your file writer class may have multiple superclasses. In general, each superclass' Init method should be invoked by your class' Init method.

---

## Error Checking

Rather than simply calling the superclass Init method, it is a good idea to check whether the call to the superclass Init method succeeded. The following statement checks the value returned by the superclass Init method; if the returned value is 0 (indicating failure), the current Init method also immediately returns with a value of 0:

```
IF ( self -> SomeFileWriterClass::Init() EQ 0 ) THEN RETURN, 0
```

This convention is used in all file writer classes included with IDL. RSI strongly suggests that you include similar checks in your own class definition files.

## Keywords to the Init Method

Properties of the file writer class can be set in the Init method by specifying the property names and values as IDL keyword-value pairs. In addition to any keywords implemented directly in the Init method of the superclass on which you base your class, the properties of the IDLitWriter class, IDLitComponent class, and IDLitMessaging class are available to any file writer class. See [“IDLitReader Properties”](#), [“IDLitComponent Properties”](#), and [“IDLitMessaging Properties”](#) in the *IDL Reference Guide* manual.

Use keyword inheritance (the `_EXTRA` keyword) to pass keyword parameters through to the superclass as necessary. (See [“Keyword Inheritance”](#) in Chapter 4 of the *Building IDL Applications* manual for details on IDL's keyword inheritance mechanism.)

## Standard Base Class

While you can create your new file writer class from any existing file writer class, in many cases, file writer classes you create will be subclassed directly from the base class `IDLitWriter`:

```
IF ( self -> IDLitWriter::Init(Extensions, TYPES = types, $
    _EXTRA = _extra) EQ 0) $
    THEN RETURN, 0
```

where *Extensions* is a string or array of strings specifying the filename extensions readable by your file writer and *types* is a string or array of strings specifying the iTool data types for which this writer is available. (See “[iTool Data Types](#)” on page 34 for details on iTool data types.)

### Note

---

The value of the *Extensions* argument is used only to display the proper filename filter when a File Save dialog is displayed — it is not a check for the proper filetype.

---

The `IDLitWriter` class provides the base iTool file writer functionality used in the tools created by RSI. See “[Subclassing from the IDLitWriter Class](#)” on page 199 for details.

## Return Value

If all of the routines and methods used in the `Init` method execute successfully, it should indicate successful initialization by returning 1. Other file writer classes that subclass from your file writer class may check this return value, as your routine should check the value returned by any superclass `Init` methods called.

## Registering Properties

File writer objects can register properties with the iTool; registered properties show up in the property sheet interface shown in the *system preferences browser* (described in “[Properties of the iTools System](#)” on page 64), and can be modified interactively by users. The iTool property interface is described in detail in [Chapter 4, “Property Management”](#).

Register a property by calling the `RegisterProperty` method of the `IDLitComponent` class:

```
self -> RegisterProperty, PropertyIdentifier [, TypeCode] $
    [, ATTRIBUTE = value]
```



where *PropertyIdentifier* is a string that uniquely identifies the property, *TypeCode* is an integer between 0 and 9 specifying the property data type, and *ATTRIBUTE* is a property attribute. See [“Registering Properties”](#) on page 54 for details.

---

### Note

A file writer need not register any properties at all, if the write operation is simple. Many of the standard iTool image file writer work without registering any properties.

---

## Setting Property Attributes

If a property has already been registered, perhaps by a superclass of your file writer class, you can change the registered attribute values using the `SetPropertyAttribute` method of the `IDLitComponent` class:

```
self -> SetPropertyAttribute, Identifier
```

where *Identifier* is the name of the keyword to the `GetProperty` and `SetProperty` methods used to retrieve or change the value of this property. (The Identifier is specified in the call to `RegisterProperty` either via the *PropertyName* argument or the `IDENTIFIER` keyword.) See [“Property Attributes”](#) on page 58 for additional details.

## Passing Through Caller-Supplied Property Settings

If you have included the `_REF_EXTRA` keyword in your function definition, you can use IDL’s keyword inheritance mechanism to pass any “extra” keyword values included in the call to the `Init` method through to other routines. One of the things this allows you to do is specify property settings when the `Init` method is called; simply include each property’s keyword/value pair when calling the `Init` method, and include the following in the body of the `Init` method:

```
IF (N_ELEMENTS(_extra) GT 0) THEN $
  self -> MyWriter::SetProperty, _EXTRA = _extra
```

where *MyWriter* is the name of your file writer class. This line has the effect of passing any “extra” keyword values to your file writer class’ `SetProperty` method, where they can either be handled directly or passed through to the `SetProperty` methods of the superclasses of your class. See [“Creating a SetProperty Method”](#) on page 196 for details.

## Example Init Method

```
FUNCTION ExampleWriter::Init, _EXTRA = _extra

  IF (self -> IDLitWriter::Init('ppm', TYPE='IDLIMAGE', $
    NAME='Portable Pixmap (PPM) File', $
```

```

        DESCRIPTION="PPM File Writer", $
        _EXTRA = _extra) EQ 0) THEN $
        RETURN, 0

    RETURN, 1

END

```

## Discussion

The `ExampleWriter` class is based on the `IDLitWriter` class (discussed in “[Subclassing from the IDLitWriter Class](#)” on page 199). As a result, all of the standard features of an `iTool` file writer class are already present. We don’t define any keyword values to be handled explicitly in the `Init` method, but we do use the keyword inheritance mechanism to pass keyword values through to methods called within the `Init` method. The `ExampleWriter` `Init` method does the following things:

1. Calls the `Init` method of the superclass, `IDLitWriter`. We specify a list of accepted filename extensions (only `ppm`, in this case) via the *Extensions* argument, and set the `TYPES` keyword. We include a description of the writer via the `DESCRIPTION` keyword. Finally, we use the `_EXTRA` keyword inheritance mechanism to pass through any keywords provided when the `ExampleWriter` `Init` method is called.
2. Returns the integer 1, indicating successful initialization.

## Creating a Cleanup Method

The file writer class `Cleanup` method handles any cleanup required by the file writer object, and should do the following:

- destroy any pointers or objects created by the file writer
- call the superclass’ `Cleanup` method

Calling the superclass’ `cleanup` method will destroy any objects created when the superclass was initialized.

### Note

---

If your file writer class is based on the `IDLitWriter` class, and does not create any pointers or objects of its own, the `Cleanup` method is not strictly required. It is always safest, however, to create a `Cleanup` method that calls the superclass’ `Cleanup` method.

---

See “[IDLitWriter::Cleanup](#)” in the *IDL Reference Guide* manual for additional details.

## Example Cleanup Method

```
PRO ExampleWriter::Cleanup

    ;; Cleanup superclass
    self -> IDLitWriter::Cleanup

END
```

### Discussion

Since our file writer object does not have any instance data of its own, the Cleanup method simply calls the superclass Cleanup method.

## Creating a GetProperty Method

The file writer class GetProperty method retrieves property values from the file writer object instance or from instance data of other associated objects. It should retrieve the requested property value, either from the file writer object's instance data or by calling another class' GetProperty method.

### Note

---

Any property registered with a call to the RegisterProperty method must be listed as a keyword to the GetProperty method either of the visualization class or one of its superclasses.

---

### Note

---

A file writer need not register any properties at all, if the write operation is simple. Many of the standard iTool image file writer work without registering any properties.

---

See [“IDLitWriter::GetProperty”](#) in the *IDL Reference Guide* manual for additional details.

## Example GetProperty Method

```
PRO ExampleWriter::GetProperty, _REF_EXTRA = _extra

    IF (N_ELEMENTS(_extra) GT 0) THEN $
        self -> IDLitWriter::GetProperty, _EXTRA = _extra

END
```

## Discussion

The `GetProperty` method first defines the keywords it will accept. There must be a keyword for each property of the file writer. Since the file writer we are creating has no properties of its own, there are no keywords explicitly defined. Note the use of the keyword inheritance mechanism to allow us to get properties from the `ExampleWriter` class' superclasses without knowing the names of the properties.

Since our `ExampleWriter` class has no properties of its own, we simply call the superclass' `GetProperty` method, passing in all of the keywords stored in the `_EXTRA` structure.

## Creating a SetProperty Method

The file writer `SetProperty` method stores property values in the file writer object's instance data. It should set the specified property value, either by storing the value directly in the visualization object's instance data or by calling another class' `SetProperty` method.

### Note

Any property registered with a call to the `RegisterProperty` method must be listed as a keyword to the `SetProperty` method either of the visualization class or one of its superclasses.

### Note

A file writer need not register any properties at all, if the write operation is simple. Many of the standard iTool image file writer work without registering any properties.

See "`IDLitWriter::SetProperty`" in the *IDL Reference Guide* manual for additional details.

## Example SetProperty Method

```
PRO ExampleWriter::SetProperty, _EXTRA = _extra

    IF (N_ELEMENTS(_extra) GT 0) THEN $
        self -> IDLitWriter::SetProperty, _EXTRA = _extra

END
```

## Discussion

The `SetProperty` method first defines the keywords it will accept. There must be a keyword for each property of the visualization type. Since the file writer we are

creating has no properties of its own, there are no keywords explicitly defined. Note the use of the keyword inheritance mechanism to allow us to set properties from the `ExampleWriter` class' superclasses without knowing the names of the properties.

Using the `N_ELEMENTS` function, we check to see whether any properties were specified via the keyword inheritance mechanism. If any keywords were specified, we call the superclass' `SetProperty` method, passing in all of the keywords stored in the `_EXTRA` structure.

## Creating a SetData Method

The file writer `SetData` method does the work of the file writer, extracting data from the selected `iTool` data object and writing the data to a file using some method. If the process is successful, the `SetData` method must return 1 for success.

In our example, we write the selected data to a Portable Pixmap (PPM) file. As a result, we do some additional checking to ensure that the data that the user has selected can be displayed as an image.

See [“IDLitWriter::SetData”](#) in the *IDL Reference Guide* manual for additional details.

### Example SetData Method

```

FUNCTION ExampleWriter::SetData, oImageData

    ; Prompt user for a file in which to save the data
    strFilename = self -> GetFilename()
    IF (strFilename EQ '') THEN $
        RETURN, 0 ; failure

    ; Check validity of the input data object
    IF (~ OBJ_VALID(oImageData)) THEN BEGIN
        self -> ErrorMessage, ['Invalid image data object'], $
            TITLE = 'Error', SEVERITY = 2
        RETURN, 0 ; failure
    ENDIF

    ; Check the iTool data type of the selected data object.
    ; If the data is not of a type that can be written to an
    ; image file, display an error message.
    oData = oImageData -> GetByType("IDLIMAGE", COUNT = count)
    IF (count EQ 0) THEN $                ;; no image, image pixels?
        oData = oImageData -> GetByType("IDLIMAGEPIXELS", $
            COUNT = count)
    IF (count EQ 0) THEN $                ;; no image, array 2d?
        oData = oImageData -> GetByType("IDLARRAY2D", COUNT = count)

```

```

IF (count EQ 0) THEN BEGIN
    self -> ErrorMessage, $
        ["Invalid data provided to file writer."], $
        TITLE="Error", SEVERITY = 2
    RETURN, 0 ; failure
END

; Turn a 1-D object array into a scalar object.
oData = oData[0]

; Determine whether the data is an image.
isImage = obj_isa(oData, "IDLitDataIDLImage")

; If data is an image, get image pixels, otherwise
; turn data into an image.
IF (isImage NE 0) THEN BEGIN
    result = oData -> GetData(image, 'ImagePixels')
ENDIF ELSE BEGIN
    result = oData -> GetData(image)
ENDELSE

; Check the result of the GetData method.
IF (result EQ 0) THEN BEGIN
    self -> ErrorMessage, ['Error retrieving image data'], $
        TITLE = 'Error', SEVERITY = 2
    RETURN, 0 ; failure
ENDIF

; Get number of dimensions of image array.
ndim = SIZE(image, /N_DIMENSIONS)

; Write to a PPM file. Use REVERSE to make image appear
; with correct orientation.
WRITE_PPM, strFilename, REVERSE(image, ndim)

; Return 1 for success.
RETURN, 1

END

```

## Discussion

The `SetData` method accepts an `IDLitData` object (`oImageData`) as its input parameter. Before processing the input data, the method prompts the user for a file in which to save the image, using the `GetFilename` method of the `IDLitWriter` object.

After securing a filename, the method proceeds to check the input data object. First it checks to make sure that the input object is valid. Then it attempts to retrieve data of an appropriate `iTool` data type from the data object; in this example, the method tries

to extract an data of one of the following types using the `GetByType` method of the `IDLitData` class:

- `IDLIMAGE`
- `IDLIMAGEPIXELS`
- `IDLARRAY2D`

If no data of any of these types is found, the method displays an error message and exits.

Once the method has obtained an appropriate data object, it checks to determine whether the data object is an `IDLitDataIDLImage` object; if so, it attempts to retrieve the image pixels from the data object; otherwise it simply retrieves the data array. The data retrieved by the `GetData` method is stored in the variable `image`. The method then checks the return value from the `GetData` method to determine whether the returned value is valid.

Using the valid image data, the method determines the number of dimensions and then uses the `WRITE_PPM` procedure to create an image file. The image data must be processed by the `REVERSE` function in order to make it appear in the output file with the correct orientation.

## Creating the Class Structure Definition

When any IDL object is created, IDL looks for an IDL class structure definition that specifies the instance data fields needed by an instance of the object, along with the data types of those fields. The object class structure must have been defined *before* any objects of the type are created. In practice, when the IDL `OBJ_NEW` function attempts to create an instance of a specified object class, it executes a procedure named `ObjectClass__define` (where *ObjectClass* is the name of the object), which is expected to define an IDL structure variable with the correct name and structure fields. For additional information on how IDL creates object instances, see “[The Object Lifecycle](#)” in Chapter 22 of the *Building IDL Applications* manual.

## Subclassing from the IDLitWriter Class

The `IDLitWriter` class is the base class for all *iTool* file writers. In almost all cases, new file will be subclassed either from the `IDLitWriter` class or from a class that is a subclass of `IDLitWriter`.

See “[IDLitWriter](#)” in the *IDL Reference Guide* manual for details on the methods properties available to classes that subclass from `IDLitWriter`.

## Example Class Structure Definition

The following is the class structure definition for the `ExampleWriter` file writer class. This procedure should be the last procedure in a file named `examplewriter__define.pro`.

```
PRO ExampleWriter__Define

    struct = { ExampleWriter,          $
              INHERITS IDLitWriter $
            }

END
```

### Discussion

The purpose of the structure definition routine is to define a named IDL structure with structure fields that will contain the visualization object instance data. The structure name should be the same as the visualization's class name — in this case, `ExampleWriter`.

Like many iTool file writer classes, `ExampleWriter` is created as a subclass of the `IDLitWriter` class. File writer classes that subclass from `IDLitWriter` class inherit all of the standard iTool file writer features, as described in [“Subclassing from the IDLitWriter Class”](#) on page 199.

The `ExampleWriter` class has no instance data of its own. For a more complex example, see [“Example: TIFF File Writer”](#) on page 203.



# Registering a File Writer

Before a file writer can be used by an iTool to write a file, the file writer's class definition must be registered as being available to the iTool. Registering a file writer with the iTool links the class definition file that contains the actual IDL code that defines the file writer with a simple string that names the writer. Code that calls a file writer in an iTool uses the name string to specify which writer should be created.

## Using IDLitTool::RegisterFileWriter

In most cases, you will register a file writer with the iTool in the iTool's class Init method. Registration ensures that the file writer is available when the iTool attempts to use it to write a file. (See “[Creating a New iTool Class](#)” on page 69 for details on the iTool class Init method.)

To register a file writer, call the IDLitTool::RegisterFileWriter method:

```
self -> RegisterFileWriter, Writer_Type, WriterType_Class_Name, $  
ICON = icon
```

where *Writer\_Type* is the string you will use when referring to the file writer, *WriterType\_Class\_Name* is a string that specifies the name of the class file that contains the file writer's definition, and *icon* is a string containing the name of a bitmap file to be used in the preferences browser.

---

### Note

The file *WriterType\_Class\_Name\_\_define.pro* must exist somewhere in IDL's path for the file writer to be successfully registered.

---

See “[IDLitTool::RegisterFileWriter](#)” in the *IDL Reference Guide* manual for details.

## Specifying Useful Properties

You can set any property of the [IDLitWriter](#) and [IDLitComponent](#) classes when registering a file writer. The following properties may be of particular interest:

### ICON

Set this property to a string value giving the name of an icon to be associated with this object. Typically, this property is the name of a bitmap file to be used when displaying the object in a tree view. See “[Icon Bitmaps](#)” on page 28 for details on where bitmap icon files are located.

# Unregistering a File Writer

If you are creating a new `iTool` from an existing `iTool` class, you may want to remove a file writer registered for the existing class from your new tool. This can be useful if you have an `iTool` class that implements all of the functionality you need, but which registers a file writer you don't want included in your `iTool`. Rather than recreating the `iTool` class to remove the file writer, you could create your new `iTool` class in such a way that it inherits from the existing `iTool` class, but *unregisters* the unwanted file writer.

Unregister a file writer by calling the `IDLitTool::UnregisterFileWriter` method in the `Init` method of your `iTool` class:

```
self -> UnregisterFileWriter, identifier
```

where *identifier* is the string name used when registering the file writer.

For example, suppose you are creating a new `iTool` that subclasses from a standard `iTool` that is based on the `IDLitToolbase` class. If you wanted your new tool to behave just like a standard tool, with the exception that it would not export PNG files, you could include the following method call in your `iTool`'s `Init` method:

```
self -> UnregisterFileWriter, 'PNG File Writer'
```

## Finding the Identifier String

To find the string value used as the *identifier* parameter to the `UnregisterFileWriter` method, you must inspect the class file that registers the file writer. In the case of our example, you would inspect the file `idlittoolbase__define.pro` to find the following call to the `RegisterFileWriter` method:

```
self -> RegisterFileWriter, 'PNG File Writer', 'IDLitReadPNG'
```

The first argument to the `RegisterFileWriter` method (`'PNG File Writer'`) is the string name of the file writer.

## Example: TIFF File Writer

This example creates a file writer to write TIFF format files. The TIFF file writer is included in the file `idlitwritetiff__define.pro`, located in the IDL distribution in the `lib/itools/components` subdirectory of the main IDL directory.

### Class Definition File

The class definition for `idlitwritetiff` consists of an `Init` method, a `SetData` method, and a class structure definition routine. As with all object class definition files, the class structure definition routine is the last routine in the file, and the file is given the same name as the class definition routine (with the suffix `.pro` appended).

#### Init Method

```
FUNCTION IDLitWriteTIFF::Init, _EXTRA = _extra

    IF (self -> IDLitWriter::Init('tiff', TYPES="IDLIMAGE", $
        NAME="Tag Image File Format", $
        DESCRIPTION="Tag Image File Format (TIFF)", $
        _EXTRA = _extra) EQ 0) THEN $
        RETURN, 0

    RETURN, 1

END
```

#### Discussion

The first item in our class definition file is the `Init` method. The `Init` method's function signature is defined first, using the class name `IDLitWriteTIFF`. Note the use of the `_EXTRA` keyword inheritance mechanism; this allows any keywords specified in a call to the `Init` method to be passed through to routines that are called within the `Init` method even if we do not know the names of those keywords in advance.

Next, we call the `Init` method of the superclass. In this case, we are creating a subclass of the `IDLitWriter` class; this provides us with all of the standard `iTool` file writer functionality automatically. Any “extra” keywords specified in the call to our `Init` method are passed to the `IDLitWriter::Init` method via the keyword inheritance mechanism.

We specify a list of accepted filename extensions (`tiff`, in this case) via the *Extensions* argument, and set the `TYPES` keyword equal to the `iTool` data type of data that can be written using this file writer. (The `iTool` data types specified by the

TYPES keyword must match the iTool data type of the data selected in the iTool Export Wizard in order for the file writer to be available for selection.)

We specify a value for the NAME property of the writer object (this is displayed in the system preferences dialog) and include a description of the writer via the DESCRIPTION keyword. Finally, we use the \_EXTRA keyword inheritance mechanism to pass through any keywords provided when the Init method is called.

Finally, we return the value 1 to indicate successful initialization.

## SetData Method

```

FUNCTION IDLitWriteTIFF::SetData, oImageData

    strFilename = self -> GetFilename()
    IF (strFilename EQ '') THEN $
        RETURN, 0 ; failure

    IF (~ OBJ_VALID(oImageData)) THEN BEGIN
        MESSAGE, "Invalid image data object.", /CONTINUE
        RETURN, 0 ; failure
    ENDIF

    result = oImageData -> GetData(image, 'ImagePixels')

    if (result eq 0) then begin
        MESSAGE, "Error retrieving image data.", /CONTINUE
        return, 0 ; failure
    endif

    ndim = SIZE(image, /N_DIMENSIONS)
    CASE ndim Of

        2: Begin ; color indexed
            success = oImageData -> GetData(palette, 'Palette')
            ; Check if we have palette data.
            IF (N_ELEMENTS(palette) GT 0) THEN BEGIN
                red = REFORM(palette[0,*])
                green = REFORM(palette[1,*])
                blue = REFORM(palette[2,*])
            ENDIF
        END

        3: BEGIN
            dims = SIZE(image, /DIMENSIONS)
            ; If we have more than 3 channels, just keep
            ; the first 3 (assumed to be RGB).
            IF (dims[0] NE 3) THEN $
                image = image[0:2, *, *]
    
```

```

        END

        ELSE: RETURN, 0 ; failure

    ENDCASE

    ; The REVERSE ensures that other applications will read in
    ; the image right side up.
    WRITE_TIFF, strFilename, REVERSE(image, ndim), $
        RED = red, GREEN = green, BLUE = blue

    RETURN, 1 ; success

END

```

## Discussion

The `SetData` method accepts an `IDLitData` object (`oImageData`) as its input parameter. Before processing the input data, the method prompts the user for a file in which to save the image, using the `GetFilename` method of the `IDLitWriter` object.

After securing a filename, the method proceeds to check the input data object. First it checks to make sure that the input object is valid. Then it attempts to retrieve data of the `iTool` data type `ImagePixels` from the data object, using the `GetData` method, storing the result in the variable `image`. The method then checks the return value from the `GetData` method to determine whether the returned value is valid.

Using the valid image data, the method determines the number of dimensions. If the image array has two dimensions, the method checks the original input data object for the presence of a palette. If the palette is present, the red, green, and blue vectors are reformed for later use by the `WRITE_TIFF` routine.

If the image array has three dimensions, it the dimensions are assumed to be the red, green, and blue channels.

Finally, the method uses the `WRITE_TIFF` procedure to create an image file. The image data must be processed by the `REVERSE` function in order to make it appear in the output file with the correct orientation.

## Class Definition

```

PRO IDLitWriteTIFF__Define

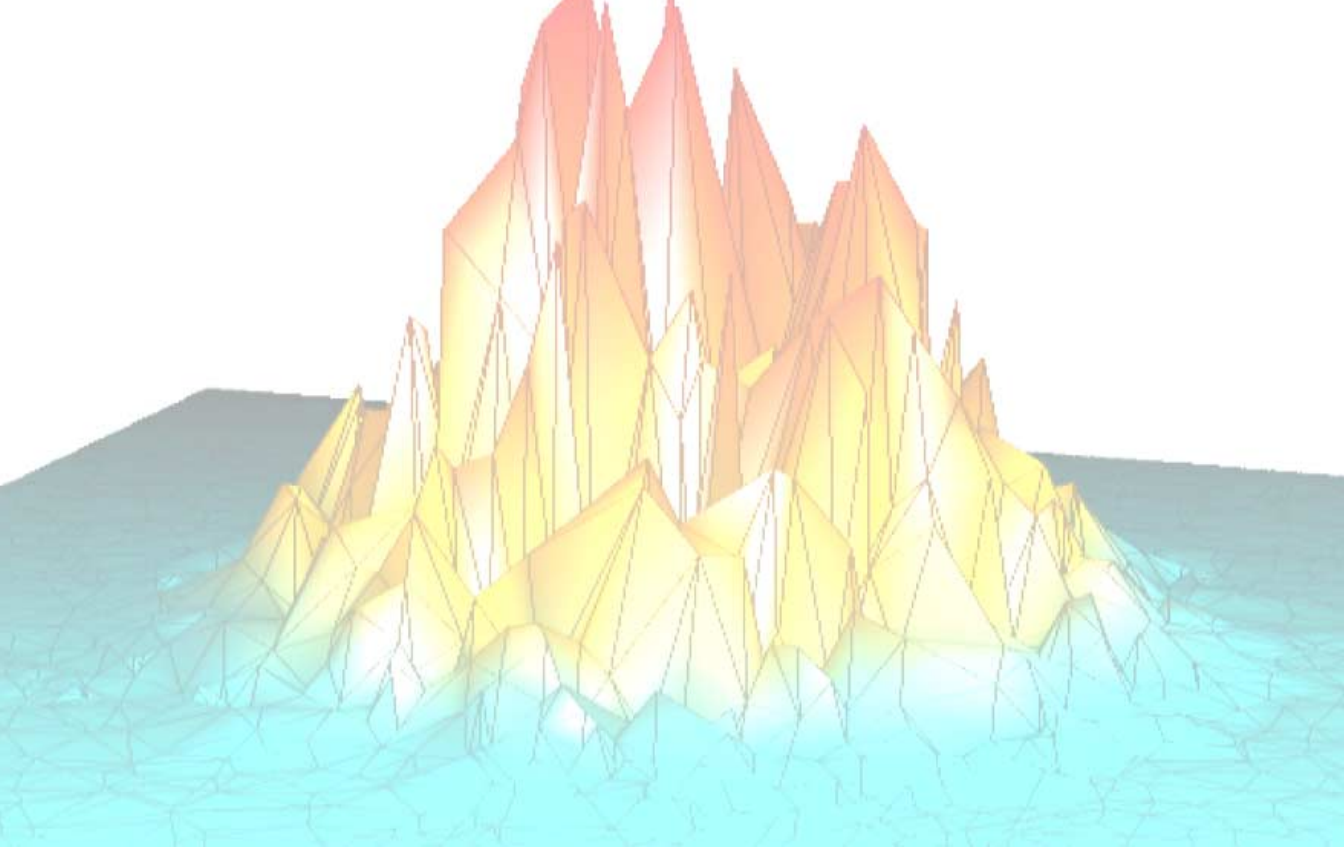
    struct = {IDLitWriteTIFF,      $
              inherits IDLitWriter $
            }

END

```

## Discussion

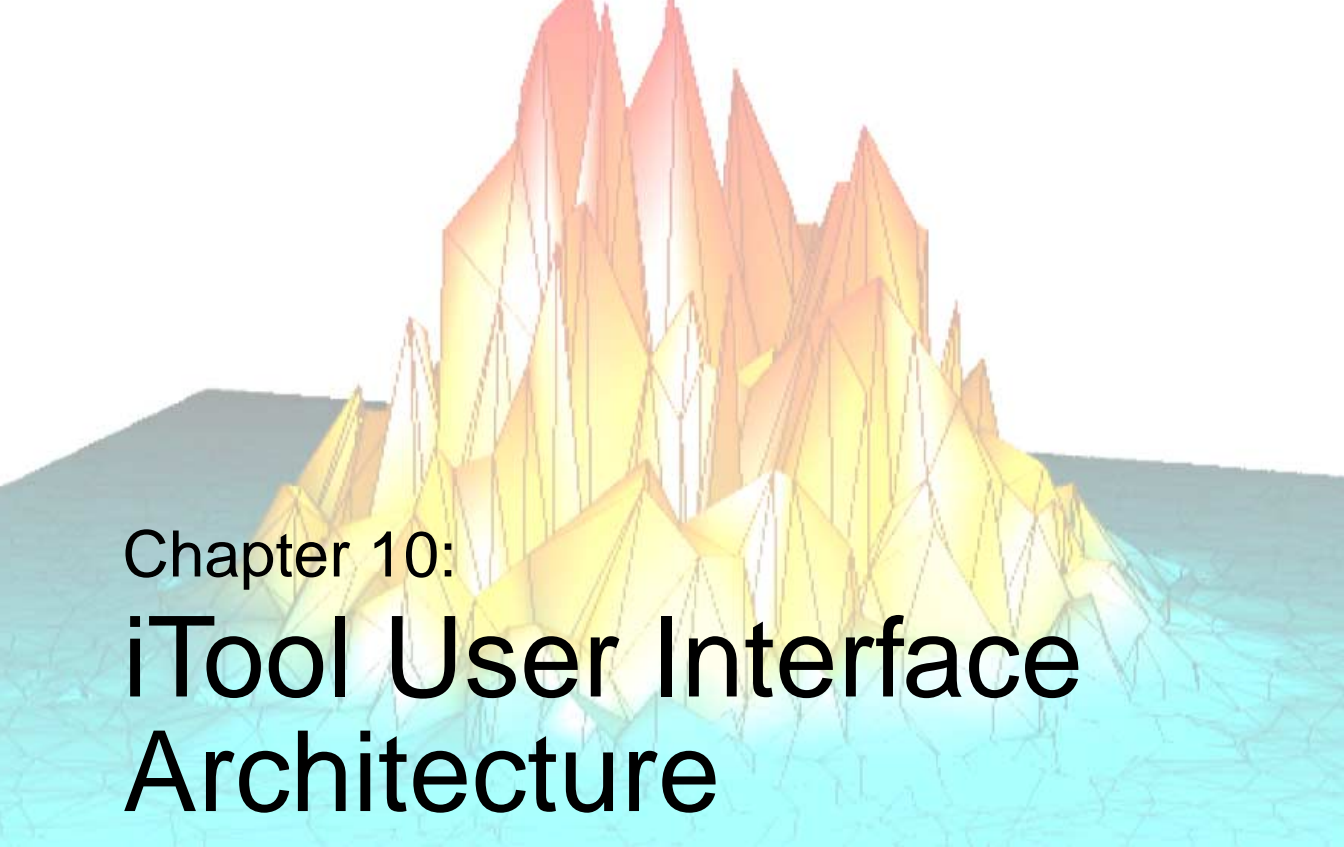
Our class definition routine is very simple. We create an IDL structure variable with the name `IDLitWriteTIFF`, specifying that the structure inherits from the `IDLitWriter` class. The object has no instance data, and thus no instance data fields.



# ***Part III: Modifying the iTool User Interface***







# Chapter 10: iTool User Interface Architecture

This chapter provides an overview of the iTool user interface architecture.

---

<a href="#">Overview</a> .....	210	<a href="#">User Interface Objects</a> .....	212
--------------------------------	-----	--	-----

# Overview

The iTool user interface architecture is designed to preserve the separation between the functionality provided by an iTool application and the manner in which that functionality is presented to the user. While the process of creating a user interface for the iTool application is complex, the idea is simple: the iTool can choose from any number of *user interface styles* that present information to the user in unique ways, depending on the operating environment.

While the initial release of the iTool component framework includes only one user interface style, created from IDL's graphical widget interface toolkit, the iTool framework design allows for the creation of additional user interface styles. Creating new interface elements, or even an entirely new user interface, does not require alterations to the underlying iTool implementation.

---

**Note**

In the first release of the IDL iTools system, the functionality necessary to create entirely new user interface styles is not fully defined. Future versions of the iTool system will provide the capability to create additional user interface styles.

---

Working within an existing interface style, you can add several different types of user interface elements to your iTools. In rough order of increasing complexity of implementation, iTool user interface elements include:

- Simple additional interface elements such as custom messages that appear in the iTool status area, informational dialogs, and simple yes-or-no type interactive user dialogs. These items can be added to an iTool using built-in methods of the `IDLitIMessaging` class. Built-in interface elements are described in [Chapter 11, “Using iTool User Interface Elements”](#).
- Modal dialogs that allow the user to provide complex information before an action is performed by the iTool. Dialog-based interface elements can be simple, perhaps allowing the user to enter a single numerical value, or complex, as shown by the iTool Curve Fitting operation's parameter-specification dialog. Dialog-based interfaces require the creation of a *user interface service*, which can then call code that creates the appropriate dialog interface for the platform and iTool interface style. User interface services are described in [Chapter 12, “Creating a User Interface Service”](#).
- iTool *panels*, which are non-modal collections of interface elements that are attached to the iTool visualization window. Panels are useful when complex controls must always be visible alongside a visualization; the iVolume and

iImage tools provide examples of a panel interface. Panel interfaces are described in [Chapter 13, “Creating a User Interface Panel”](#).

# User Interface Objects

The iTool user interface object is an instance of the class `IDLitUI`. The UI object provides a way for the iTool to communicate with interface elements created using the IDL widget toolkit. As the center of communication between the user interface and the underlying iTool functionality, the UI object provides the following functionality:

- Access to and communication with the underlying iTool object.
- Registration and management of dialogs and other sub-elements of the user interface that are used by the iTool to perform specific tasks.
- Registration of user interface elements that are part of the iTool display itself.

One of the key features of the iTool user interface is the ability to adapt to the contents of the tool, sensitizing and desensitizing menu items or displaying dialogs or user interface panels as necessary. The `IDLitUI` object makes this adaptability possible while maintaining the slender link between tool functionality and user interface. The following features of the `IDLitUI` object make these features possible:

## GetTool Method

The `IDLitUI::GetTool` method provides the means to retrieve an object reference to the underlying iTool object from user interface code. The retrieved reference can then be used to access data stored in iTool objects (property values, for example) and to call other iTool object methods.

## UI Service Registration Methods

The `IDLitUI::RegisterUIService` and `IDLitUI::UnRegisterUIService` methods allow user interface code to register (and unregister) user interface services as being available for use by the iTool interface.

---

### Note

User interface services are more normally registered by an iTool launch routine, using the `ITREGISTER` procedure.

---

User interface services are discussed in detail in [Chapter 12, “Creating a User Interface Service”](#).

## Widget Registration Methods

The [IDLitUI::RegisterWidget](#) and [IDLitUI::UnRegisterWidget](#) methods allow user interface code to register (and unregister) widget callback routines as the target of `OnNotify` messages. Registration allows the user interface to receive messages generated by iTool components and to react accordingly.

Widget registration is discussed in detail in [Chapter 13, “Creating a User Interface Panel”](#).

## AddOnNotifyObserver Method

The [IDLitUI::AddOnNotifyObserver](#) method allows user interface code to register to receive messages sent via calls to the `OnNotify` methods of iTool components. This mechanism allows the user interface to change in response to changes in the underlying iTool.

Use of the iTool messaging system is discussed in detail in [Chapter 13, “Creating a User Interface Panel”](#).

## DoAction Method

The [IDLitUI::DoAction](#) method makes it possible for a user interface element to launch execution of an operation within the underlying iTool.

Use of the `DoAction` method to initiate execution of operations is discussed in [Chapter 12, “Creating a User Interface Service”](#).





# Chapter 11: Using iTool User Interface Elements

This chapter describes user interface elements that can be incorporated into an iTool without the need to write any user interface code.

---

<a href="#">Overview</a> .....	216	<a href="#">Prompts</a> .....	219
<a href="#">Status Messages</a> .....	217	<a href="#">Informational Messages</a> .....	221

## Overview

The IDLitIMessaging class provides methods that allow you to accept and return feedback via the iTool interface without writing any interface code yourself. For many applications, adding the ability to provide status information, prompt the user for simple input, and display appropriate error messages to the standard iTool interface is sufficient; in these cases, no additional code is needed to create and display user interfaces.

---

**Note**

The simple dialogs presented by the IDLitIMessaging methods are similar to those displayed by the IDL DIALOG\_MESSAGE function. Since the initial iTools release supports only one user interface style (built using the IDL widget interface toolkit) it may be tempting to use DIALOG\_MESSAGE rather than the methods described in this chapter. As the iTools framework matures, however, additional user interface styles may be created either by RSI or by third-party developers. Using the built-in IDLitIMessaging methods will ensure that your iTool applications continue to function properly when other interface styles are available.

---

This chapter discusses the use of the basic user interface elements provided by the IDLitIMessaging class. If your application requires a more complex interface, see [Chapter 12, “Creating a User Interface Service”](#) or [Chapter 13, “Creating a User Interface Panel”](#).



# Status Messages

*Status messages* are simple text messages displayed in a way that does not impede the user's operation of the iTool. In the standard iTool user interface created using the IDL widget toolkit, status messages are text strings displayed at the bottom of the iTool window.

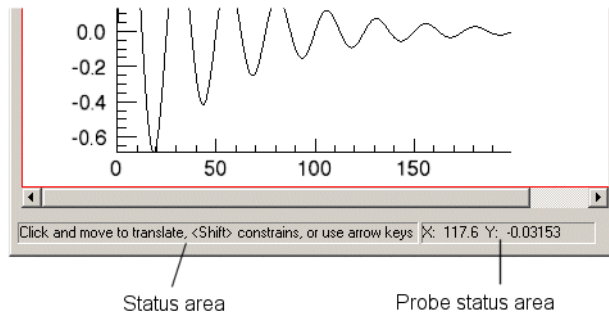


Figure 11-1: The status areas of a standard iTool.

The `IDLitIMessaging` class provides two methods that display status messages. See “[IDLitIMessaging](#)” in the *IDL Reference Guide* manual for details.

## StatusMessage

The `IDLitIMessaging::StatusMessage` method displays a string value. In the standard iTool interface created using the IDL widget toolkit, status messages appear in the *status area* at the bottom left corner of the iTool window, as shown in [Figure 11-1](#).

In the standard set of iTools provided with IDL, the status area is used to display status information for operations or informational messages pertaining to the currently selected object or manipulator.

The following code places the text “My Status Message” in the status area:

```
self -> StatusMessage, 'My Status Message'
```

## ProbeStatusMessage

The `IDLitIMessaging::ProbeStatusMessage` method displays a string value. In the standard iTool interface created using the IDL widget toolkit, probe status messages appear at the bottom right corner of the iTool window, as shown in [Figure 11-1](#).

In the standard set of iTools provided with IDL, the probe status area is used to display the position of the cursor within the iTool window.

The following code places the text “X: 300, Y:146” in the status area:

```
self -> ProbeStatusMessage, 'X: 300, Y:146'
```

In most cases, the values displayed in the probe status area have some relationship to the position of the cursor or to the action performed by the current manipulator.

# Prompts

*Prompts* solicit information from the user. Prompts are generally presented as modal dialogs, meaning that the user must respond to the prompt before operation of the iTool can continue.

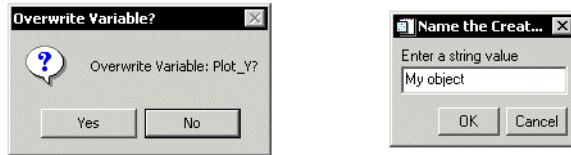


Figure 11-2: Yes/No and Text Prompt dialogs.

The `IDLitIMessaging` class provides two methods that prompt for user input. See “[IDLitIMessaging](#)” in the *IDL Reference Guide* manual for details.

## PromptUserYesNo

The `IDLitIMessaging::PromptUserYesNo` method displays a prompt string along with **Yes** and **No** buttons. In the standard iTool interface created using the IDL widget toolkit, Yes/No prompts appear as modal dialogs as shown in [Figure 11-2](#).

### Note

The `PromptUserYesNo` function returns 1 if the dialog executed properly. You must check the value stored in the variable specified as the *Answer* argument to determine which button the user pressed.

The following code asks the user a Yes or No question and performs some action if the dialog returns properly *and* the value of the returned variable `answer` is equal to 1 (as would be the case if the user clicked **Yes**):

```
status = self -> PromptUserYesNo('Overwrite Variable: Plot_Y', $
    answer, TITLE='Overwrite Variable?')

IF (status NE 0 && answer EQ 1) THEN BEGIN
    ; do something...
ENDIF
```

The value of the `TITLE` keyword is displayed in the title bar of the dialog box.

## PromptUserText

The `IDLitMessaging::PromptUserText` method displays a prompt string and a text-entry field along with **OK** and **Cancel** buttons. In the standard iTool interface created using the IDL widget toolkit, text prompts appear as modal dialogs as shown in [Figure 11-2](#).

---

### Note

The `PromptUserText` function returns 1 if the user clicks the **OK** button, or 0 if the user clicks the **Cancel** button.

---

The following code asks the user to enter a text string, which will be stored in the variable `stringName`:

```
status = self -> PromptUserText('Enter a string value', $
    stringName, TITLE = 'Name the Created Object')
```

The value of the `TITLE` keyword is displayed in the title bar of the dialog box. The variable `status` will contain a 1 if the user clicks **OK**, or a 0 if the user clicks **Cancel**.

# Informational Messages

*Informational Messages* inform the user that some condition has occurred in the iTool application. The condition may be an error, but it can also be any other occurrence of which the user should be informed. Informational messages are presented as modal dialogs, generally with a single OK button that dismisses the dialog.

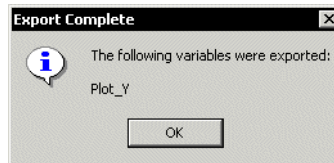


Figure 11-3: An informational message dialog.

The `IDLitMessaging` class provides the `ErrorMessage` method to display informational messages of all sorts. See “[IDLitMessaging](#)” in the *IDL Reference Guide* manual for details.

## ErrorMessage

The `IDLitMessaging::ErrorMessage` method displays an informational text message to the user. In the standard iTool interface created using the IDL widget toolkit, informational messages appear as modal dialogs as shown in [Figure 11-3](#).

Informational messages can use any of three severity codes, indicating to the user whether the message is merely informational, is a warning, or reports a serious error. While the severity setting does not alter the behavior of the dialog, which can only be dismissed by the user, it can alter the appearance of the dialog. For example, the dialog shown in [Figure 11-3](#) has a severity setting of 0, or “Informational”.

The following code displays an informational message:

```
self -> ErrorMessage, ['The following variables were exported:', $
    'Plot_Y'], SEVERITY = 0, TITLE = 'Export Complete'
```

The value of the `TITLE` keyword is displayed in the title bar of the dialog box.

In addition to the `ErrorMessage` method, the `IDLitMessaging` class provides the `SignalError` method, which reports an error condition to the iTool system but which does not display the message to the user. See “[IDLitMessaging](#)” in the *IDL Reference Guide* manual for details.





## Chapter 12: Creating a User Interface Service

This chapter describes the process of creating a user interface service.

---

<a href="#">Overview</a> .....	224	<a href="#">Registering a UI Service</a> .....	231
<a href="#">Predefined iTool UI Services</a> .....	225	<a href="#">Executing a User Interface Service</a> .....	233
<a href="#">Creating a New UI Service</a> .....	226	<a href="#">Example: Changing a Property Value</a> ...	234

# Overview

A *UI service* is an iTool component object class that defines how and when a user interface element is presented to an iTool user. UI services provide a way to separate platform-independent iTool functionality from platform-dependent user interface code. When an iTool needs to display a graphical interface, it simply calls the appropriate UI service to display the interface; the iTool itself does not need to know anything at all about the platform on which it is running. Decisions about how to display the desired interface are left to the UI service, which can choose from any number of options based on the platform and user interface style in use.

---

**Note**

In the initial iTools release, only one user interface style is supplied: the IDL widget interface toolkit. As the iTools framework continues to grow, additional user interface styles may be created either by RSI or by third-party developers.

---

## Creating and Using a UI Service

To create and use a new iTool UI service, you will do the following:

- Create an IDL function that displays the user interface elements. See [“Creating a New UI Service”](#) on page 226 for details.
- Register the new UI service with the iTools system. See [“Registering a UI Service”](#) on page 231 for details.
- Execute the UI service from iTool code. See [“Executing a User Interface Service”](#) on page 233 for details.



# Predefined iTool UI Services

The iTool system distributed with IDL includes a number of pre-defined UI services. These UI services are registered with the iTool system, which means that you can call them from any operation, visualization, or other iTool component using the `DoUIService` method of the `IDLitTool` class.

The majority of the pre-defined UI services provide interface elements that are specific to the standard iTool implementation. In most cases, you do not need to call these services directly; using the existing iTool operation or visualization code that calls the UI service is sufficient. If you are creating a new UI service, you may want to inspect the code for some of the standard UI services — they are located in the `lib/itools/ui_widgets` subdirectory of the IDL directory and have file names of the form `idlitui*.pro`.

The following UI services are generally useful; you may wish to include calls to these services in your own iTool operation or visualization code.

## Hourglass Cursor Service

Displays the hourglass cursor. The hourglass cursor is displayed until processing completes and a new IDL widget event is processed, at which time the previous cursor is reinstated.

### File Name

`idlituihourglass.pro`

### Registered Service Name

`HourGlassCursor`

### Example

```
void = oTool -> DoUIService('HourGlassCursor', self)
```

# Creating a New UI Service

A user interface service is responsible for creating a user interface element that is displayed when an iTool user takes some action. A simple UI service may do no more than display the “hourglass” cursor while an operation is being performed; more complicated UI services may be small applications unto themselves.

For simple operations the UI service routine can contain everything necessary to implement the UI service. For more complex interfaces, however, it is often practical to separate the actual user interface code (that is, the widget creation and event-handling routines) from the logic of the UI service itself. The latter is the strategy used by many of the UI services included with the standard iTools.

The process of creating a user interface service is outlined in the following sections:

- [“Creating the UI Service Routine”](#) on page 226
- [“Creating Supporting User Interface Elements”](#) on page 229

## Creating the UI Service Routine

The user interface service routine performs the following tasks:

- Manages changes to any properties of the object on which the user interface element was invoked.
- Manages the display of the user interface element.

To accomplish these things, the UI service routine needs a reference to the iTool component on which the service will act, and a reference to the IDLitUI object associated with the current iTool. As a result, the user interface service routine has the following signature:

```
FUNCTION ServiceName, oUI, oRequester
```

where *ServiceName* is the name of the function, *oUI* is an object reference to the IDLitUI object associated with the iTool, and *oRequester* is an object reference to the iTool component specified in the call to the DoUIService method.

---

### Note

*ServiceName* is not necessarily the same as the registered name of the service used in the call to the DoUIService method. The registered name is defined by the call to the ITREGISTER procedure. See [“Registering a UI Service”](#) on page 231 for details.

---

## Return Value

The user interface service routine should return 1 if the action succeeds, or 0 otherwise.

## Retrieving Property Information

The *oRequester* argument to the user interface service function contains an object reference to the *iTool* component on which the UI service was invoked. Use this reference to retrieve any properties of the object that are relevant to the operation being performed by the user interface.

For example, the standard `ScaleFactor` user interface service displays a dialog that lets the user set the `SCALE_FACTOR` property of an object. The service uses the following statement to retrieve the current scale factor from the selected object:

```
oRequester -> GetProperty, SCALE_FACTOR = factor
```

## Retrieving Widget Information

The *oUI* argument to the user interface service function contains an object reference to the `IDLitUI` object associated with the current *iTool*. You can use this reference to retrieve the IDL widget identifier of the widget that is the *group leader* of the *iTool* user interface itself (the *iTool* window); the ID is stored in the `GROUP_LEADER` property of the `IDLitUI` object. Having this widget ID allows you to retrieve screen geometry information that allows you to calculate the position at which your user interface should be displayed.

For example, the `ScaleFactor` user interface service uses the following code to calculate the X and Y offsets that will be used to position its own user interface over the current *iTool*:

```
; Retrieve the widget ID of top-level base.
oUI -> GetProperty, GROUP_LEADER = groupLeader

IF (WIDGET_INFO(groupLeader, /VALID)) THEN BEGIN
    geom = WIDGET_INFO(groupLeader, /GEOMETRY)
    xoffset = geom.scr_xsize + geom.xoffset - 80
    yoffset = geom.yoffset + (geom.ysize - 400)/2
ENDIF
```

The UI service goes on to use the calculated `xoffset` and `yoffset` values when positioning the IDL widgets that make up the interface displayed by the service.

## Displaying the User Interface

If the user interface being displayed by the UI service is simple, it may be convenient to include the code for creating it directly in the definition of the user interface service itself. For example, the following is the complete definition of the `HourGlassCursor` user interface service:

```
FUNCTION IDLitUIHourGlass, oUI, oRequester
    WIDGET_CONTROL, /HOURGLASS
    RETURN, 1
END
```

As you can see, no information about the `IDLitUI` object or the selected `iTool` component is used, and the displayed item itself is very simple.

In most cases, the user interface service is significantly more complex. In these cases it is often useful to separate the routine that creates the service's user interface from the code that displays it. For example, the user interface for the `ScaleFactor` service is displayed by the following statement:

```
result = IDLitwdScaleFactor(GROUP_LEADER = groupLeader, $
    FACTOR = factor, XOFFSET = xoffset, YOFFSET = yoffset)
IF result EQ 1 THEN RETURN, 0
```

This statement calls another function — `IDLitwdScaleFactor` — to actually display the required user interface elements, supplying the information retrieved by other portions of the user interface service routine. The `IDLitwdScaleFactor` function returns the scale factor value selected by the user, or returns the value 1 (indicating no scaling) if the value supplied by the user is invalid. If the returned scale factor is 1 (either because the user entered 1 the value, or because the entered value was not a valid value), no scaling will be performed, so the UI service itself returns the failure value (integer 0). The process of creating user interface elements is discussed in greater detail in [“Creating Supporting User Interface Elements”](#) on page 229.

## Setting Property Information

If the user has selected new values for any of the object's properties, those properties values must be changed on the object by a call to the `SetProperty` method. In our example, if the user sets a new scale factor, the following statement updates the property value, notifies the selected object that the value has changed, and inserts the change into the undo-redo transaction buffer:

```
oRequester -> SetProperty, SCALE_FACTOR = result
```

Note that not every user interface will modify properties of the selected object.

## Example

The following example routine is the full definition of the ScaleFactor user interface service described in the previous sections. It is presented here again for completeness, so you can see the entire function at once.

```

FUNCTION IDLituiScaleFactor, oUI, oRequester

    ; Retrieve widget ID of top-level base.
    oUI -> GetProperty, GROUP_LEADER = groupLeader

    ; Retrieve geometry information and calculate offsets.
    IF (WIDGET_INFO(groupLeader, /VALID)) THEN BEGIN
        screenSize = GET_SCREEN_SIZE(RESOLUTION = resolution)
        geom = WIDGET_INFO(groupLeader, /GEOMETRY)
        xoffset = geom.scr_xsize + geom.xoffset - 80
        yoffset = geom.yoffset + (geom.ysize - 400)/2
    ENDIF

    ; Retrieve the current scale factor from the selected object.
    oRequester -> GetProperty, SCALE_FACTOR = factor

    ; Display the IDL widget interface allowing the user to
    ; change the scale factor. The new scale factor is returned
    ; as the result of this function. If the specified value is
    ; not a valid scale factor, the integer 1 is returned in
    ; result.
    result = IDLitwdScaleFactor( GROUP_LEADER = groupLeader, $
        FACTOR = factor, XOFFSET = xoffset, YOFFSET = yoffset)
    IF result EQ 1 THEN RETURN, 0

    ; Set properties on the selected object.
    oRequester -> SetProperty, SCALE_FACTOR = result

    ; Return success.
    RETURN, 1

END

```

## Creating Supporting User Interface Elements

It is beyond the scope of this manual to provide general information on the creation of user interfaces. For information on creating a user interface using the IDL widget toolkit, see [“Creating Graphical User Interfaces in IDL”](#) in the *Building IDL Applications* manual. The following are some suggestions for creating IDL widget interface code for iTool user interface services.

**Place data collected by the user interface in the function's return value**

Create your user interface routine (the routine that creates the IDL widgets that make up the user interface displayed by your UI service) as a function, returning the data values collected by the interface in the function's return value. If you are collecting several values of different data types, return a structure variable containing the data. The user interface and event-handling code should never change data or property values within the *iTool* itself; all changes should be made via the `SetProperty` mechanism

**Be sure to clean up heap variables when the user interface exits**

If your user interface code creates pointer or object heap variables, be sure to destroy them before the interface code exits. If extra “hanging” heap variables are left undestroyed, IDL can potentially run out of resources if the interface is displayed numerous times.

**Use the `GROUP_LEADER` property if it is available**

Pass the widget ID contained in the `GROUP_LEADER` property of the `IDLitUI` object to your user interface code, and set the `GROUP_LEADER` keyword of the top-level base widget to this value. Setting the widget group leader to the leader of the *iTool*'s own widget hierarchy ensures that your user interface will be destroyed if the *iTool* itself is destroyed.

# Registering a UI Service

Before a user interface service can be called from an iTool, the routine that implements the service must be registered with the iTool system. Registering a UI service with the system links the file containing the actual IDL code that creates the user interface elements with a simple string that names the UI service. Since you use the name string in code that calls the service, the iTool itself does not need to know anything about the display environment in which it is running.

User interface services are registered either using the ITREGISTER procedure or via a call to the RegisterUIService method of the IDLitUI object. In most cases, registration is accomplished via a call to the ITREGISTER procedure in an iTool's launch routine. A UI service can be registered at any time. In practice, you will probably find it convenient to register UI services used by an iTool in the iTool launch routine, unless you know the service has already been registered. For a list of UI services that are pre-registered by the standard iTools, see [“Predefined iTool UI Services”](#) on page 225.

## Using ITREGISTER

Use the ITREGISTER routine to register a user interface service:

```
ITREGISTER, 'UI Service Name', 'UI_Service_Routine', /UI_SERVICE
```

where *UI Service Name* is a string you will use to call the user interface service, and *UI\_Service\_Routine* is a string that specifies the name of the file that contains the code for the user interface service.

### Note

---

The file *UI\_Service\_Name\_\_define.pro* must exist somewhere in IDL's path for the service definition to be successfully registered.

---

If a given user interface service has already been registered when the ITREGISTER routine is called, the service will not be registered a second time. The registration can be performed at any time in an IDL session before you attempt to call the user interface service.

See [“ITREGISTER”](#) in the *IDL Reference Guide* manual for details.

## Example

Suppose you have a UI service definition file named `myUIService.pro`, located in a directory included in IDL's `!PATH` system variable. Register this service with the `iTool` system with the following command:

```
ITREGISTER, 'My UI Service', 'myUIService', /UI_SERVICE
```

The user interface service can now be invoked via the `DoUIService` method:

```
success = oTool -> DoUIService('My UI Service', self)
```

where `oTool` is an object reference to the current `iTool` object.

## Using the RegisterUIService Method

User interface services can also be registered by a call to the `RegisterUIService` method of the `IDLitUI` object:

```
self -> RegisterUIService, 'My UI Service', 'myUIService'
```

---

**Note**

In most cases, you do not have a reference to the `IDLitUI` object available, so this method is not generally useful. We mention it here because the user interface services registered for use by the standard `iTools` are registered in this way, rather than via the `ITREGISTER` procedure.

---



## Executing a User Interface Service

Once you have defined and registered a user interface service and created any supporting user interface code, you can call the service from any `iTool` operation simply by calling the `DoUIService` method of the `IDLitTool` class.

In most cases, the `DoUIService` method is called from the `DoExecuteUI` method of an `IDLitOperation` or an `IDLitDataOperation`. For example, the following routine is the `DoExecuteUI` method of an operation that calls the `ScaleFactor` user interface service:

```
FUNCTION IDLitopScalefactor::DoExecuteUI

    oTool = self -> GetTool()
    IF (oTool EQ OBJ_NEW()) THEN RETURN, 0

    RETURN, oTool -> DoUIService('ScaleFactor', self)

END
```

The `GetTool` method is part of the `IDLitIMessaging` class, which is a superclass of all `iTool` operation classes; it returns an object reference to the current `iTool`. This method calls the `ScaleFactor` user interface service with the operation itself as the currently selected object, which allows the UI service to modify the operation's properties. The second argument to the `DoUIService` method is an object reference that can be used by the service to modify the object's properties.

## Example: Changing a Property Value

This example creates a user interface service named `SrvExample`, which displays a dialog that allows the user to change the `NAME` property of the currently selected `iTool` component. The `SrvExample` user interface service is launched by an `IDLitDataOperation` named `opName`.

This example is intended as a demonstration of the techniques used to create a user interface service. In practice, you do not have to create a user interface to change the `NAME` property; it can be changed more easily by altering the value in the Visualization browser. It is conceivable, however, that you might want to provide an interface that allows the user to change numerous properties simultaneously, with some values being based on other user-supplied values. Similarly, you may wish to display a dialog that allows the user to set the properties of an operation every time that operation is executed, without forcing the user to open the Operations browser.

Creating and using the `SrvExample` user interface service involves the following steps:

- [Creating the `SrvExample` service](#)
- [Creating the `SrvExample` interface](#)
- [Creating an operation that calls the service](#)
- [Registering the `SrvExample` service](#)
- [Registering the `opName` operation](#)
- [Invoking the `opName` operation](#)

### Creating the `SrvExample` service

The `SrvExample` user interface service consists of a single function named `SrvExample`, stored in a file named `srvexample.pro` that is located in a directory that is included in the `IDL PATH` system variable.

```
FUNCTION SrvExample, oUI, oRequester

    ; Retrieve widget ID of top-level base.
    oUI -> GetProperty, GROUP_LEADER = groupLeader

    ; Retrieve the original value of the name property
    ; attribute from the selected item.
    oRequester -> GetProperty, NAME = origName
```

```

; Display the widget UI that allows the user to choose
; a new name.
newName = wdSrvExample(NAME = origName, $
    GROUP_LEADER = groupLeader)

; Set the property value.
oRequester -> SetProperty, NAME = newName

; Return success
RETURN, 1

END

```

## Discussion

The function that implements this example service follows the pattern outlined in [“Creating the UI Service Routine”](#) on page 226. It uses the object reference to the IDLitUI object to retrieve the widget ID of the top-level base of the iTool user interface, and later uses the retrieved value to set the GROUP\_LEADER keyword to the user interface routine. It uses the object reference to the “requester” object (in this case, the iTool component that is selected in the current iTool) to retrieve the NAME property. It then calls a routine (wdSrvExample) that displays a user interface allowing the user to select a new value for the NAME property.

The string returned by the wdSrvExample routine is used to set the NAME property of the selected iTool component, and the routine returns 1 for success.

## Creating the SrvExample interface

The interface presented by the SrvExample user interface service consists of a set of routines that create an IDL widget interface. The creation routine and two simple event-handling routines are stored in a file named wdsrvexample.pro that is located in a directory that is included in the IDL PATH system variable.

### Widget Creation Function

The following function creates the widget interface that is displayed when the SrvExample user interface service is called. The widget creation routine should be the last routine in the file.

```

FUNCTION wdSrvExample, NAME = origName, TITLE = dialogTitle, $
    GROUP_LEADER = groupLeader

; Check to see if a title for the dialog was supplied.
; If not, set a default title.
IF (N_ELEMENTS(dialogTitle) EQ 0) THEN $
    dialogTitle='Choose a Name'

```

```

; Create the dialog.
wBase = WIDGET_BASE(/COLUMN, TITLE = dialogTitle, $
    GROUP_LEADER = groupLeader)
wText = WIDGET_TEXT(wBase, YSIZE = 3, $
    VALUE=['The original NAME is:', origName, $
        'Enter a new name:'])
wEdit = WIDGET_TEXT(wBase, VALUE = origName, /EDITABLE)
wSubBase = WIDGET_BASE(wBase, /ROW)
wOK = WIDGET_BUTTON(wSubBase, VALUE='OK', $
    EVENT_PRO='wdSrvExample_ok')
wCancel = WIDGET_BUTTON(wSubBase, VALUE='Cancel', $
    EVENT_PRO='wdSrvExample_cancel')

; Create a state structure to hold important values.
state = { wOK:wOK, $
    wCancel:wCancel, $
    wEdit:wEdit, $
    pName:PTR_NEW(/ALLOCATE) }

; Store the original property name attribute in the
; state structure.
*state.pName = origName

; Store the state structure in the user value of the
; top-level widget base.
WIDGET_CONTROL, wBase, SET_UVALUE = state

; Realize the widget hierarchy.
WIDGET_CONTROL, wBase, /REALIZE

; Call XMANAGER.
XMANAGER, 'wdSrvExample', wBase

; After XMANAGER exits, retrieve the value of the name
; property attribute from the state structure.
result = (N_ELEMENTS(*state.pName)) ? *state.pName : origName

; Free the pointer.
PTR_FREE, state.pName

; Return the new value of the name property attribute.
RETURN, result

```

END

## Discussion

It is beyond the scope of this chapter to discuss the IDL widget programming techniques used in this example. For more information on widget programming, see the *Building IDL Applications* manual. Several points are worth noting, however.

- The widget ID of the top-level base retrieved in the `SrvExample` routine is passed to this routine, and used as the value of the `GROUP_LEADER` keyword to `WIDGET_BASE`. This ensures that if the `iTool` itself is minimized or closed while the example dialog is displayed, it will be handled properly.
- The original value of the `NAME` property is passed to this routine, and is stored in an IDL pointer variable within the state structure that is associated with the dialog. This allows the event routine that actually retrieves the value entered by the user to communicate the new value back to the widget creation routine, but it also means that the pointer must be freed before the routine exits.

## Event-handling Routines

The following event-handling procedures handle widget events generated by the widget interface that is displayed when the `SrvExample` user interface service is called.

```

PRO wdSrvExample_ok, event

    ; Get the stashed state structure from the user value
    ; of the top-level base widget.
    WIDGET_CONTROL, event.top, GET_UVALUE = state

    ; Get the value from the editable text field.
    WIDGET_CONTROL, state.wEdit, GET_VALUE = value

    ; Store the text value in a pointer so we can access
    ; it from the main routine
    *state.pName = value

    ; Destroy the dialog.
    WIDGET_CONTROL, event.top, /DESTROY

END

PRO wdSrvExample_cancel, event

    ; Nothing to do, just destroy the dialog.
    WIDGET_CONTROL, event.top, /DESTROY

END

```

## Discussion

When the user clicks the OK button, the current value of the editable text widget is placed in the pointer stored in the state structure's `pName` field.

## Creating an operation that calls the service

In order to launch the `SrvExample` user interface service, the user must be able to select an operation that calls the `DoUIService` method. This example uses an `IDLitDataOperation` named `opName`, which simply retrieves the list of currently selected items and calls the `SrvExample` user interface service. The code for this operation is stored in a file named `opname__define.pro` that is located in a directory that is included in the `IDL PATH` system variable.

```

FUNCTION opName::Init, _EXTRA = _extra

    ; Initialize the operation, setting the "show UI" property.
    ; Note that this operation will operation on all iTool
    ; component types.
    success = self -> IDLitDataOperation::Init( $
        NAME="Rename Component", $
        DESCRIPTION="Rename an iTool component", $
        /SHOW_EXECUTION_UI, TYPES='')

    RETURN, success

END

FUNCTION opName::DoExecuteUI

    ; Get a reference to the current iTool and
    ; make sure it is valid.
    oTool = self -> GetTool()
    IF (oTool eq OBJ_NEW()) THEN RETURN, 0

    ; Get the list of selected items.
    selItem = oTool -> GetSelectedItems()

    ; Call the UI service on the first item in the list
    ; of selected items.
    RETURN, oTool -> DoUIService('Example Service', selItem[0])

END

```

```
PRO opName__define

    struct = {opName, $
              inherits IDLitDataOperation $
            }

END
```

## Discussion

Only two methods are required: `Init` and `DoExecuteUI`. Since this operation is based on the `IDLitDataOperation` class, all interaction with the `iTools` undo/redo system is automated.

Even though all of the items that are currently selected in the `iTool` are retrieved by the `GetSelectedItems` method, only the first item is passed to the `SrvExample` user interface service for processing. Handling multiple selected items would require a more complicated user interface.

The process of defining an `IDLitDataOperation` is discussed in detail in [Chapter 7](#), “[Creating an Operation](#)”.

## Registering the `SrvExample` service

In order for the `SrvExample` user interface service to be available, it must be registered with the current `iTool`. The following line in the `iTool`’s launch routine allows the service to be called with the name “`Example Service`”:

```
ITREGISTER, 'Example Service', 'srvExample', /UI_SERVICE
```

## Registering the `opName` operation

To use the `opName` operation within an `iTool`, the operation must be registered in the `iTool`’s definition. The following statement registers the operation with the name “`Property Name`” and places it in the `Operations` menu of the `iTool`.

```
self -> RegisterOperation, 'Property Name', 'opName', $
    IDENTIFIER = 'Operations/PropertyName'
```

## Invoking the `opName` operation

To use the `SrvExample` service, the user would launch an `iTool` for which the `opName` operation is registered, select an `iTool` component in the window, and select **Property Name** from the **Operations** menu.







# Chapter 13: Creating a User Interface Panel

This chapter describes the process of creating a user interface panel.

---

<a href="#">Overview</a> .....	242	<a href="#">Registering a UI Panel</a> .....	250
<a href="#">Creating a UI Panel Interface</a> .....	243	<a href="#">Example: A Simple UI Panel</a> .....	252
<a href="#">Creating Callback Routines</a> .....	248		

# Overview

A *UI Panel* is a collection of user interface elements displayed in one or more tabs located on the right, left, or bottom edge of an iTool window. The UI panel interface makes it easy to attach a set of controls chosen by the iTool developer to the standard iTool interface.

---

**Note**

In the initial iTools release, only one user interface style is supplied: the IDL widget interface toolkit. As a result, UI panels consist of widgets from the IDL graphical user interface toolkit, displayed in a tab widget. As the iTools framework continues to grow, additional user interface styles may be created either by RSI or by third-party developers.

---

Controls on a UI panel exchange information with the iTool itself via one or more *callback routines*. These routines allow the iTool to modify the controls in the UI panel as the user selects different visualization components or otherwise changes the contents of the visualization.

## Creating and Using a UI Panel

To add a UI panel to the iTool interface, you will do the following:

- Create an IDL procedure that creates the user interface elements that comprise the panel. See [“Creating a UI Panel Interface”](#) on page 243 for details.
- Create an one or more event-handling routines to handle events generated by the user interface elements in the panel. See [“Creating a UI Panel Interface”](#) on page 243 for details.
- Create one or more callback routines to control the display of the items on the panel as the contents of the iTool window change. See [“Creating Callback Routines”](#) on page 248 for details.
- Create an iTool with the TYPES property set to the appropriate iTool type and register the UI panel with the iTool that will display it. See [“Registering a UI Panel”](#) on page 250 for details.

# Creating a UI Panel Interface

It is beyond the scope of this manual to provide general information on the creation of user interfaces. For information on creating a user interface using the IDL widget toolkit, see “[Creating Graphical User Interfaces in IDL](#)” in the *Building IDL Applications* manual. Keep the following points in mind when creating IDL widget interface code for iTool user interface panels.

## Panel Creation Routines

A user interface panel creation routine is similar to the widget creation routine that creates a standalone widget application, but with the following important differences:

### Signature

The routine signature of a user interface panel looks like this:

```
PRO PanelName, wPanel, oUI
```

where *PanelName* is the name of the routine, *wPanel* is an input argument that contains the widget ID of the panel widget associated with this panel, and *oUI* is an input argument that contains an object reference to the IDLitUI object associated with the iTool that includes the user interface panel.

### Event Loop and Widget Management

Standalone widget applications must arrange for the *management* of their widgets and the creation of an *event loop*; these details are usually handled by the XMANAGER or WIDGET\_EVENT routines. A user interface panel does not need to call XMANAGER or WIDGET\_EVENT; widget management is handled by the main iTool interface code. A user interface panel simply attaches itself to the bulk of the iTool interface.

## About the Panel Widget

In the initial release of the iTools, user interface panels are contained in an IDL tab widget displayed on the right side of the iTool window. We will refer to this tab widget as the *panel widget* in this documentation, since all user interface elements in a UI panel are contained in this widget.

The panel widget itself is created automatically when a user interface panel is registered with an iTool, and its widget ID is passed to the panel creation routine along with a reference to the iTool user interface object.

Use the widget ID of the panel widget to set the title of the tab that appears at the top of the panel. For example the following lines might occur at the beginning of a routine that builds a user interface panel:

```
PRO ExamplePanel, wPanel, oUI

    ; Set the title used on the panel's tab.
    WIDGET_CONTROL, wPanel, BASE_SET_TITLE='Example Panel'

    ... more panel code.
```

The `wPanel` argument contains the widget ID of the panel widget, which was assigned when the `iTool` interface was built. The `oUI` argument contains an object reference to the `IDLitUI` object associated with the current `iTool`. The call to the `WIDGET_CONTROL` procedure sets the title of the tab to be “Example Panel.”

You may also find it useful to specify a single event-handling routine for all events generated by the panel widget. You can specify the name of this routine with a statement similar to the following:

```
WIDGET_CONTROL, wPanel, EVENT_PRO = 'ExamplePanel_event'
```

where `ExamplePanel_event` is replaced by the name of the event-handling routine you create for your panel. Of course, you can also specify event-handling routines for specific widgets within the panel using the `EVENT_PRO` and `EVENT_FUNC` keywords to the widget creation routines.

## Registering the Panel with the User Interface Object

To ensure that notifications from the `iTool` itself are passed to the user interface panel as needed, the panel creation routine must register the panel widget with the `iTool` user interface object. This registration step allows you to specify the name of the *callback routine* that will be called when a notification is generated by the `iTool` itself.

To register a user interface panel, use the `RegisterWidget` method of the `IDLitUI` object:

```
id = oUI -> RegisterWidget(wPanel, 'Panel', 'Ex_callback')
```

where `oUI` is an object reference to the `IDLitUI` object and `wPanel` is the widget ID of the panel widget; both are passed in as arguments to the panel creation routine. The second argument to the `RegisterWidget` method (`'Panel'`, in this example) is the human-readable name of the UI panel. The third argument (`'Ex_callback'`, in this example) is the name of the panel’s callback routine. See “[IDLitUI::RegisterWidget](#)” in the *IDL Reference Guide* manual for details. Callback routines are discussed in detail in “[Creating Callback Routines](#)” on page 248.

## Adding Observers

For notification messages to be passed to the correct callback routine, an `OnNotifyObserver` must be established by calling the `AddOnNotifyObserver` method of the `IDLitUI` object. The `AddOnNotifyObserver` method takes as its arguments the ID created by the call to the `RegisterWidget` method (as discussed in the previous section) and the component object identifier of the `iTool` component to observe. Once the observer is created, each time the specified `iTool` component generates a message (that is, when the component itself calls the `DoOnNotify` method), the registered widget callback routine is called with the message as one of its arguments. The call to the `AddOnNotifyObserver` method looks like:

```
oUI -> AddOnNotifyObserver, id, Component
```

where *id* is an identifier created by a call to the `RegisterWidget` method, and *Component* is the component object identifier of the `iTool` component being observed. See “[IDLitUI::AddOnNotifyObserver](#)” in the *IDL Reference Guide* manual for additional details.

The *component* argument to the `AddOnNotifyObserver` method can be any string value. For example, any time the selection within an `iTool` window changes, the `DoOnNotify` method is called with its first parameter (*idOriginator*) set to the string value 'Visualization' rather than to the object identifier of a component. An observer whose *Component* argument is set to the string 'Visualization' will be notified each time the selection changes in the `iTool` window. For example, the following statement specifies that the panel widget (as registered via the `RegisterWidget` method) will receive notifications whenever a visualization changes in the `iTool` window.

```
oUI -> AddOnNotifyObserver, id, 'Visualization'
```

Here, *id* is the identifier created in the previous section. The second argument ('Visualization') specifies that messages will be generated whenever a visualization is modified.

“[Example: A Simple UI Panel](#)” on page 252 provides examples of observers of both types. See “[iTool Messaging System](#)” on page 25 for background information on observers and messages.

## Create the Widget Hierarchy

The widget hierarchy of a user interface panel looks like the following:

```

Panel widget
  |
  - Base widget
    |
    - other widgets
  
```

Since the widget ID of the panel widget is supplied as an argument to the panel creation routine, all that is left is to create a base widget with the panel widget as its parent, and to populate the base widgets with other widgets as necessary.

### Passing State Information

State information can be passed between widget creation routines and widget event handling routines in several different ways. The method used most often in iTool user interface panels is to create a state structure in the panel creation routine, store the appropriate values in this structure, and assign the structure to the widget user value of one of the widgets in the panel widget hierarchy. For a more detailed discussion of this technique, see [“Managing Application State”](#) in Chapter 26 of the *Building IDL Applications* manual.

In addition to widget IDs and other state information from your widget interface, you may find it useful to store object references to the iTool object and to the IDLitUI object associated with the iTool object in the state structure. Having these object references available in your event handler and callback routines allows you to take advantage of methods available in the iTool and user interface objects.

### Create Event Handlers

Like other widget applications, iTool user interface panels use one or more event handling routines to perform actions based on the user’s interaction with the widgets in the interface. As with generalized widget applications, you can write event handling routines for a user interface panel in numerous ways; see [“Widget Event Processing”](#) in Chapter 26 of the *Building IDL Applications* manual for an in-depth discussion of widget event handling in general.

The following suggestions apply specifically to event handlers for iTool user interface panels:

## Use the GetSelectedItems Method

Often, you will want to apply an operation to one or more items in the iTool window when the user selects an element on the user interface panel. Use the `GetSelectedItems` method of the iTool object to retrieve references to the iTool component objects that are selected.

The following statement retrieves an array of object references to all of the currently selected items in the iTool:

```
oTargets = state.oTool -> GetSelectedItems(COUNT = nTarg)
```

---

### Note

Note that this example assumes that a reference to the iTool object is stored in the `oTool` field of the `state` structure variable. The `COUNT` keyword to the `GetSelectedItems` method returns the number of items selected.

---

## Use the DoAction Method

In many cases, the user's interaction with the user interface panel will instruct the iTool to apply an iTool operation to the selected item. Where possible, use the `DoAction` method of the operation to perform this task. Calling the `DoAction` method ensures that the changes caused by the operation are properly inserted into the iTool undo/redo system.

For example, the following statement:

```
success = state.oUI -> DoAction('Operations/Rotate/RotateLeft')
```

calls the `DoAction` method on the `IDLitUI` object associated with the current iTool, invoking the operation registered with the system with the operation identifier `'Operations/Rotate/RotateLeft'`.

## Redraw the iTool Window

Call the `RefreshCurrentWindow` method of the iTool object to force the iTool's window to update, displaying any changes that took place as the result of the operations executed in your event handling routine:

```
state.oTool -> RefreshCurrentWindow
```

---

### Note

Note that this example assumes that a reference to the iTool object is stored in the `oTool` field of the `state` structure variable.

---

# Creating Callback Routines

User interface panel callback routines are executed when an iTool component for which the panel has created an *observer* generates a *notification message*. The callback routine then uses the value of the notification message to determine what action to take. Observers are created as described in [“Adding Observers”](#) on page 245.

## Callback Routine Signature

A user interface panel widget callback routine has the following signature:

```
PRO PanelName_callback, wPanel, IdOriginator, IdMessage, Value
```

where:

- *PanelName\_callback* is the name of the callback routine,
- *wPanel* is the widget ID of the panel widget (see [“About the Panel Widget”](#) on page 243),
- *IdOriginator* is a string identifying the source of the message (usually the object identifier of an iTool component object, but it can be any string value),
- *IdMessage* is a string that uniquely identifies the message being sent, and
- *Value* is a value that is associated with the message being sent.

See [“iTool Messaging System”](#) on page 25 for more information on the *IdMessage* and *Value* arguments.

## Registration of Callback Routines

Callback routines are registered along with the user interface panel itself, in the call to the RegisterWidget method of the IDLitUI object. See [“Registering the Panel with the User Interface Object”](#) on page 244 for details.

## Retrieving Widget State Information

The *wPanel* argument to the callback routine contains the widget ID of the panel widget. This widget ID provides a way for the callback routine to retrieve state information about the widgets that make up the panel.



For example, if you have saved a state structure containing widget information in the user value of the first child widget of the panel widget, code similar to the following would allow you to retrieve that state structure:

```
; Make sure we have a valid widget ID.  
IF ~ WIDGET_INFO(wPanel, /VALID) THEN RETURN  
  
; Retrieve the widget ID of the first child widget of  
; the UI panel.  
wChild = WIDGET_INFO(wPanel, /CHILD)  
  
; Retrieve the state structure from the user value of  
; the first child widget.  
WIDGET_CONTROL, wChild, GET_UVALUE = state
```

This technique is used in the example user interface panel described in [“Example: A Simple UI Panel”](#) on page 252.

# Registering a UI Panel

User interface panels are registered with the iTool system using the `ITREGISTER` procedure. Once a UI panel has been registered, it will be displayed for any iTool whose `TYPE` property matches the string specified via the `TYPES` keyword when registering the panel. Similarly, if an iTool displays a visualization whose `TYPE` property matches the string specified via the `TYPES` keyword when registering the panel, the panel will be displayed for that iTool.

## Registering the Panel in the iTool Launch Routine

In most cases, you will register your user interface panel in an iTool's launch routine, with a statement like:

```
ITREGISTER, panelName, panelCode, TYPES = panelType, /UI_PANEL
```

where *panelName* is a string containing the human-readable name of your user interface panel, *panelCode* is a string containing the name of the IDL procedure that creates the user interface panel, and *panelType* is a string that identifies the type of iTool or visualization for which the panel should be displayed. The `UI_PANEL` keyword must be present in order to register a user interface panel using the `ITREGISTER` procedure.

See “[ITREGISTER](#)” in the *IDL Reference Guide* manual for additional details.

## About the TYPE property

To display a user interface panel for a given iTool, you will not only need to register the panel in that iTool's launch routine, but also specify a matching type when initializing the iTool itself. The iTool system will display a registered panel in an iTool whose `TYPE` property contains a string that matches the string specified via the `TYPES` keyword when registering the panel.

To set the `TYPE` property of an iTool use a statement like this in the iTool's `Init` method:

```
self -> IDLitToolbase::Init(_EXTRA = _extra, TYPE = panelType)
```

where *panelType* is a string that matches the string used as the value of the `TYPES` keyword to `ITREGISTER`.

Similarly, the iTool system will display a registered panel when an iTool displays a visualization whose `TYPE` property contains a string that matches the string specified via the `TYPES` keyword when registering the panel.

To set the `TYPE` property of a visualization, use a statement like this in the visualization's `Init` method:

```
self -> IDLitVisualization::Init(_EXTRA = _extra, TYPE = panelType)
```

where *panelType* is a string that matches the string used as the value of the `TYPES` keyword to `ITREGISTER`.

## Changing the Panel Location

You can control which side of the iTool the user interface panel is displayed on by specifying the `PANEL_LOCATION` keyword to the `IDLITSYS_CREATETOOL` function. The keyword can be set to any of the following values;

- 0 = position the panel above the iTool window
- 1 = position the panel below the iTool window
- 2 = position the panel to the left of the iTool window.
- 3 = position the panel to the right of the iTool window (this is the default).

### Note

---

If your iTool creation routine uses the keyword inheritance mechanism, and the `_EXTRA` keyword is included in the creation routine's call to `IDLITSYS_CREATETOOL`, then the user will be able to specify the `PANEL_LOCATION` keyword when launching the iTool from the IDL command line.

---

## Example: A Simple UI Panel

The following example creates a simple user interface panel consisting of two buttons: Rotate and Hide/Show. The Rotate button rotates the selected iTool component 90 degrees, if possible. The Hide/Show button toggles the value of the HIDE property of the selected object.

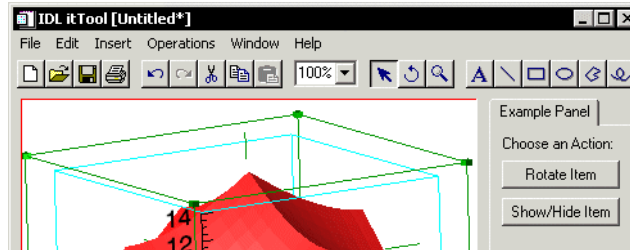


Figure 13-1: The example panel.

---

### Note

This example is intended to demonstrate the concepts involved in creating a user interface panel. For examples of more useful panels, see the files `idlitmngmenu.pro` and `idlitvolmenu.pro`, which create the user interface panels for the IIMAGE and IVOLUME iTools, respectively. Both files are located in the `lib/itools/ui_widgets` subdirectory of the IDL installation directory.

---

To display a user interface panel named *ExamplePanel*, this example creates the following items:

- [Panel Creation Routine](#)
- [Panel Event Handler Routine](#)
- [Panel Callback Routine](#)
- [Panel Type Specification](#)

## Panel Creation Routine

The user interface panel creation routine does the work of displaying the IDL widgets that make up the UI panel display.

```

PRO ExamplePanel, wPanel, oUI

; Set the title used on the panel's tab.
WIDGET_CONTROL, wPanel, BASE_SET_TITLE = 'Example Panel'

; Specify the event handler
WIDGET_CONTROL, wPanel, EVENT_PRO = "ExamplePanel_event"

; Register the panel with the user interface object.
strObserverIdentifier = oUI -> RegisterWidget(wPanel, "Panel", $
    'ExamplePanel_callback')
; Register to receive selection events on visualizations.
oUI -> AddOnNotifyObserver, strObserverIdentifier, $
    'Visualization'

; Retrieve a reference to the current iTool.
oTool = oUI -> GetTool()

; Create a base widget to hold the contents of the panel.
wBase = WIDGET_BASE(wPanel, /COLUMN, SPACE = 5, /ALIGN_LEFT)

; Create panel contents.
wLabel = WIDGET_LABEL(wBase, VALUE = "Choose an Action:", $
    /ALIGN_LEFT)

; Get the Operation ID of the rotate operation. If the operation
; exists, create the "Rotate Item" button and monitor whether
; the operation is available for the selected item.
opID = 'Operations/Operations/Rotate/RotateLeft'
oRotate = oTool -> GetByIdentifier(opID)

IF (OBJ_VALID(oRotate)) THEN BEGIN
    idRotate = oRotate -> GetFullIdentifier()
    wRotate = WIDGET_BUTTON(wBase, VALUE = "Rotate Item", $
        UVALUE="ROTATE")
    ; Monitor for availability of the Rotate operation.
    oUI -> AddOnNotifyObserver, strObserverIdentifier, idRotate
ENDIF ELSE $
idRotate = 0

wHide = WIDGET_BUTTON(wBase, VALUE = "Show/Hide Item", $
    UVALUE = "HIDE")

; Pack up the state structure and store in first child.
state = {oTool:oTool, $
    oUI:oUI, $
    idRotate : idRotate, $
    wPanel:wPanel, $
    wBase:wBase, $

```

```

        wRotate:wRotate, $
        wHide:wHide $
    }
wChild = WIDGET_INFO(wPanel, /CHILD)

IF wChild NE 0 THEN $
    WIDGET_CONTROL, wChild, SET_UVALUE = state, /NO_COPY

END

```

## Discussion

It is beyond the scope of this chapter to describe the IDL widget concepts employed in the `ExamplePanel` example; the comments in the code that creates the user interface panel describe most of the features. The following points are worth noting, however:

- The panel creation routine accepts two arguments: the widget ID of the panel widget (stored in the variable `wPanel`, in this example), and an object reference to the `IDLitUI` object associated with the `iTool` (stored in the variable `oUI`).
- The example uses the `EVENT_PRO` keyword to the `WIDGET_CONTROL` procedure to establish an event-handling routine, `ExamplePanel_event`. This event-handling routine is described in [“Panel Event Handler Routine”](#) on page 255.
- The example registers a single callback routine, `ExamplePanel_callback`, using the `RegisterWidget` method of the `IDLitUI` class. The callback routine is described in [“Panel Callback Routine”](#) on page 256.
- The example adds an `OnNotifyObserver` for the `Visualization` component described in [“Adding Observers”](#) on page 245.
- The example uses the `GetTool` method of the `IDLitUI` object to retrieve an object reference to the `iTool` with which the panel is associated. This reference is later used to retrieve a reference to the `IDLitOperation` object that performs the `Rotate Left` operation, placing it in the variable `oRotate`.
- If the `Rotate Left` operation is available to the `iTool`, the example places the `Rotate` button on the user interface panel. It also establishes an observer to watch for changes in the availability of the `Rotate Left` operation, which will change based on the item selected. The callback routine will use the messages received by this observer to sensitize and desensitize the `Rotate` button as necessary.

- The example packages important information in a state structure, and assigns this structure to the user value of the first child widget of the panel widget. The event-handling and callback routines will retrieve this state structure and use the information contained therein.

## Panel Event Handler Routine

The event-handler routine receives widget events generated by the widgets that make up the user interface panel, and acts accordingly.

```

PRO ExamplePanel_event, event

; Retrieve the widget ID of the first child widget of
; the UI panel.
wChild = WIDGET_INFO(event.handler, /CHILD)

; Retrieve the state structure from the user value of
; the first child widget.
WIDGET_CONTROL, wChild, GET_UVALUE = state

; Retrieve the user value of the widget that generated
; the event.
WIDGET_CONTROL, event.id, GET_UVALUE = uvalue

;; Now do the work for each panel item.
SWITCH STRUPCASE(uvalue) OF
  'ROTATE': BEGIN
    ; Apply the Rotate Left operation to the selected item.
    success = state.oUI -> DoAction(state.idRotate)
    RETURN
  END
  'HIDE': BEGIN
    ; Hide the selected item.
    ;
    oTargets = state.oTool -> GetSelectedItems(count = nTarg)
    IF nTarg GT 0 THEN BEGIN
      ; If there are selected items, use only the last
      ; selection.
      oTarget = oTargets[0]
      ; Get the iTool identifier of the selected item.
      name = oTarget -> GetFullIdentifier()
      ; Retrieve the setting of the HIDE property.
      oTarget -> GetProperty, HIDE = hide
      ; Change the value of the HIDE property from 0 to 1
      ; or from 1 to 0. Use the DoSetProperty and
      ; CommitActions method to ensure that the change
      ; is entered into the undo/redo transaction buffer.
      void = state.oTool -> DoSetProperty(name, "HIDE", $

```

```

        ((hide+1) MOD 2))
        state.oTool -> CommitActions
    ENDIF
    BREAK
END
ELSE:
ENDSWITCH

; Refresh the iTool window.
state.oTool -> RefreshCurrentWindow

END

```

## Discussion

It is beyond the scope of this chapter to describe the IDL widget concepts employed in the ExamplePanel event handler; the comments in the code describe most of the features. The following points are worth noting, however:

- If the event received by the event handler routine is generated by the `Rotate` button, the example calls the `DoAction` method of the `IDLitUI` object, with the identifier of the `Rotate Left` operation as its argument.
- If the event received by the event handler routine is generated by the `Hide/Show` button, the example does the following:
  - Use the reference to the `iTool` object stored in the state structure to retrieve the list of selected items using the `GetSelectedItems` method.
  - Retrieve the object identifier of the last item selected.
  - Retrieve the value of the `HIDE` property of the selected item.
  - Use the `DoSetProperty` method of the `IDLitTool` object to toggle the value of the `HIDE` property for the selected item.
  - Commit the property change in the undo/redo transaction buffer using the `CommitActions` method of the `IDLitTool` object.
- After the `iTool` display has been changed, call the `RefreshCurrentWindow` method of the `IDLitTool` object to redraw the `iTool` window.

## Panel Callback Routine

The user interface panel callback routine is called whenever a component for which an `OnNotifyObserver` has been registered generates a message. It parses the message received and takes action as necessary.



```

PRO ExamplePanel_callback, wPanel, strID, messageIn, component

; Make sure we have a valid widget ID.
IF ~ WIDGET_INFO(wPanel, /VALID) THEN RETURN

; Retrieve the widget ID of the first child widget of
; the UI panel.
wChild = WIDGET_INFO(wPanel, /CHILD)

; Retrieve the state structure from the user value of
; the first child widget.
WIDGET_CONTROL, wChild, GET_UVALUE = state

; Process as necessary, depending on the message received.
SWITCH STRUPCASE(messageIn) OF

; This section handles messages generated when the rotate
; operation becomes available or unavailable, and sensitizes
; or desensitizes the "Rotate" button accordingly.
'SENSITIVE':
'UNSENSITIVE': BEGIN
    WIDGET_CONTROL, state.wRotate, $
        SENSITIVE = (messageIn EQ 'SENSITIVE')
    BREAK
END

; This section handles messages generated when the
; item selected in the iTool window changes and changes
; the sensitivity of the "Hide/Show" button accordingly.
'SELECTIONCHANGED': BEGIN
    ; Retrieve the item that was selected last.
    oSel = state.oTool -> GetSelectedItems()
    oSel = oSel[0]
    ; If the last item selected is not a visualization,
    ; desensitize the "Hide/Show" button.
    IF (~OBJ_ISA(oSel, 'IDLITVISUALIZATION')) THEN $
        WIDGET_CONTROL, state.wHide, SENSITIVE = 0 $
    ELSE BEGIN
        ; If the selected object is a visualization, sensitize
        ; the "Hide/Show" button.
        WIDGET_CONTROL, state.wHide, SENSITIVE = 1
    ENDELSE
    BREAK
END
ELSE:
ENDSWITCH

END

```

## Discussion

The example panel's callback routine performs the following tasks:

- Uses the widget ID provided in the `wPanel` argument to retrieve the widget state structure stored in the first child widget of the panel widget.
- If the value of the `messageIn` argument is either `SENSITIVE` or `UNSENSITIVE`, change the sensitivity of the `Rotate` button (stored in the `wRotate` field of the widget state structure) as necessary.
- If the value of the `messageIn` argument is `SELECTIONCHANGED`, perform the following tasks:
  - Use the reference to the `iTool` object stored in the `oTool` field of the state structure to retrieve an object reference to the last selected component.
  - If the selected component is not a visualization, desensitize the `Hide/Show` button.
  - If the selected component is a visualization, sensitize the `Hide/Show` button.

## Panel Type Specification

In order to display the `ExamplePanel` user interface panel along with an `iTool`, the following two things must happen:

1. The UI panel must be registered, using the `ITREGISTER` procedure.
2. A tool with the appropriate `TYPE` must be created.

For the purposes of this example, suppose we have an `iTool` named `myTool`, with a launch routine named `myTool.pro`, and an `iTool` object definition routine named `myTool__define.pro`.

In the `myTool.pro` file, we included the following statement:

```
ITREGISTER, 'Example Panel', 'ExamplePanel', TYPE = 'EXAMPLE', $
/UI_PANEL
```

In the `myTool__define.pro` file, we include the string `EXAMPLE` in the `TYPE` property specified in the `Init` method:

```
FUNCTION myTool::Init, _REF_EXTRA = _EXTRA

IF (self -> IDLitToolbase::Init(_EXTRA = _extra, $
TYPE = 'EXAMPLE') EQ 0) $

THEN RETURN, 0
```

Calling the launch routine `myTool` at the IDL Command Line creates a new `iTool` and displays the `ExamplePanel` panel on the right side of the `iTool` window.





# Index

## *Symbols*

`_EXTRA` keyword, [81](#)

## **A**

Add method, [61](#)  
AddByIdentifier method, [33](#)  
adding data, [33](#)  
AddOnNotifyObserver method, [27](#), [213](#), [245](#)  
AGGREGATE keyword, [61](#)  
Aggregate method, [61](#)  
aggregation of properties, [50](#), [61](#)  
architecture of iTools, [17](#)  
attributes, [50](#)  
automatic data type matching, [43](#)

## **B**

base class  
  file reader, [168](#)  
  file writer, [192](#)  
  iTool, [70](#)  
  operation, [128](#), [141](#)  
  visualization, [97](#)  
bitmap location, [28](#)  
boolean properties, [51](#)  
BOOLEAN property data type, [51](#)

## **C**

callback routines  
  creating, [248](#)  
  for user interface, panel, [242](#)  
  observers, [245](#)

callback routines (*continued*)  
 registering, 248

Cleanup method

- data operation, 130
- file reader, 170
- file writer, 194
- generalized operation, 142
- visualization, 102

color properties, 51

COLOR property data type, 51

command line arguments, 80

component framework *See* framework

component registration, 22

components, 75

container

- data, 36, 37
- parameter, 37

creating

- file readers, 162, 166
- file writers, 186
- iTools, 67
- operations, 120
- user interface services, 226
- visualization types, 90, 95

## D

data

- container, 36
- management, 31
- manager
  - adding data, 33
  - described, 33
  - removing data, 33

objects

- described, 36
- IDLitDataIDLArray2D, 38
- IDLitDataIDLArray3D, 38
- IDLitDataIDLImage, 39
- IDLitDataIDLImagePixels, 39
- IDLitDataIDLPalette, 39
- IDLitDataIDLPolyvertex, 39

data (*continued*)

objects

IDLitDataIDLVector, 40

removing, 33

types

- IDLARRAY2D, 35
- IDLARRAY3D, 35
- IDLIMAGE, 35
- IDLIMAGEPIXELS, 35
- IDLOPACITY\_TABLE, 35
- IDLPALETTE, 35
- IDLPOLYVERTEX, 35
- IDLVECTOR, 35
- IDLVERTEX, 35
- iTool, 32
- matching, 43
- parameter, 32, 41
- property, 49
- property *See* property data types
- update mechanism, 45

data-centric operations, 125

DESCRIPTION property attribute, 59

DoAction method

- generalized operation, 143
- user interface element, 213

documented classes, 11

DoExecuteUI method, 132

## E

enumerated list properties, 53

ENUMLIST

- property attribute, 59
- property data type, 53

error handling, 82

ErrorMessage method, 221

examples

- data operation, 156
- file reader, 179
- file writer, 203
- simple iTool, 85
- simple user interface panel, 252

- examples (*continued*)
  - user interface service, 234
  - visualization type, 113
- Execute method
  - data operation, 131
  - described, 123
- EXPENSIVE\_OPERATION property, 123, 153

## F

- file readers
  - creating, 162, 166
  - described, 162
  - example, 179
  - IDLitReadASCII, 163
  - IDLitReadBinary, 163
  - IDLitReadBMP, 163
  - IDLitReadDICOM, 163
  - IDLitReadISV, 164
  - IDLitReadJPEG, 164
  - IDLitReadPICT, 164
  - IDLitReadPNG, 164
  - IDLitReadTIFF, 164
  - IDLitReadWAV, 165
  - predefined, 163
  - preferences, 64
  - registering, 72, 177
  - standard base class, 168
  - unregistering, 178

- file writers
  - creating, 186
  - described, 186
  - example, 203
  - IDLitWriteASCII, 187
  - IDLitWriteBinary, 187
  - IDLitWriteBMP, 187
  - IDLitWriteISV, 188
  - IDLitWriteJPEG, 188
  - IDLitWritePICT, 188
  - IDLitWritePNG, 188
  - IDLitWriteTiff, 189

- file writers (*continued*)
  - predefined, 187
  - preferences, 64
  - registering, 72, 201
  - standard base class, 192
  - unregistering, 202
- FLOAT property data type, 51
- floating-point integer properties, 51
- framework
  - advantages, 9
  - architecture, 17
  - code base, 11
  - documented vs. undocumented classes, 11
  - overview
  - skills required to use, 13

## G

- GetData method to file reader, 174
- GetProperty method
  - and property identifiers, 57
  - data operation, 133
  - file reader, 171
  - file writer, 195
  - generalized operation, 147
  - visualization, 103
- GetTool method, 212

## H

- help, 29
- HIDE property attribute, 59
- hierarchy, 21

## I

- icon (bitmap) location, 28
- ICON property, 154, 201
- IDENTIFIER
  - keyword, 80
  - property, 154

- identifiers
  - property, 50, 57
  - strings *See* object identifiers
- IDL widgets, 18, 210, 246
- IDLARRAY2D data type, 35
- IDLARRAY3D data type, 35
- IDLgr\* graphics objects, 99
- IDLIMAGE data type, 35
- IDLIMAGEPIXELS data type, 35
- IDLit\* visualization objects, 99
- IDLitData object, 36
- IDLitData objects, 33
- IDLitDataContainer object, 36
- IDLitDataContainer objects, 33
- IDLitDataIDLArray2D data object, 38
- IDLitDataIDLArray3D data object, 38
- IDLitDataIDLImage data object, 39
- IDLitDataIDLImagePixels data object, 39
- IDLitDataIDLPalette data object, 39
- IDLitDataIDLPolyvertex data object, 39
- IDLitDataIDLVector data object, 40
- IDLitDataOperation class, 128, 136
- IDLitDataOperation object, 125
- IDLitIMessaging class, 216
- IDLitIMessaging object, 25
- IDLitOpBytscl operation, 122
- IDLitOpConvolution operation, 122
- IDLitOpCurvefitting operation, 122
- IDLitOperation class, 141, 152
- IDLitOpSmooth operation, 122
- IDLitParameterSet object, 37, 81
- IDLitParameterSet objects, 33
- IDLitReadASCII file reader, 163
- IDLitReadBinary file reader, 163
- IDLitReadBMP file reader, 163
- IDLitReadDICOM file reader, 163
- IDLitReader class, 168
- IDLitReadISV file reader, 164
- IDLitReadJPEG file reader, 164
- IDLitReadPICT file reader, 164
- IDLitReadPNG file reader, 164
- IDLitReadTIFF file reader, 164
- IDLitReadWAV file reader, 165
- IDLITSYS\_CREATETOOL function, 83
- IDLitToolbase class, 70, 75
- IDLitUI class, 212
- IDLitUIHourGlass user interface service, 225
- IDLitVisAxis visualization type, 91
- IDLitVisColorbar visualization type, 91
- IDLitVisContour visualization type, 91
- IDLitVisHistogram visualization type, 91
- IDLitVisImage visualization type, 91
- IDLitVisIsosurface visualization type, 92
- IDLitVisLegend visualization type, 92
- IDLitVisLight visualization type, 92
- IDLitVisPlot visualization type, 92
- IDLitVisPlot3D visualization type, 92
- IDLitVisPolygon visualization type, 93
- IDLitVisPolyline visualization type, 93
- IDLitVisRoi visualization type, 93
- IDLitVisSurface visualization type, 93
- IDLitVisText visualization type, 93
- IDLitVisualization class, 97, 108
- IDLitVisVolume visualization type, 94
- IDLitWriteASCII file writer, 187
- IDLitWriteBinary file writer, 187
- IDLitWriteBMP file writer, 187
- IDLitWriteISV file writer, 188
- IDLitWriteJPEG file writer, 188
- IDLitWritePICT file writer, 188
- IDLitWritePNG file writer, 188
- IDLitWriter class, 192, 199
- IDLitWriteTIFF file writer, 189
- IDLOPACITY\_TABLE data type, 35
- IDLPALETTE data type, 35
- IDLPOLYVERTEX data type, 35
- IDLVECTOR data type, 35
- IDLVERTEX data type, 35
- informational messages, 221
- Init method
  - data operation, 126
  - file reader, 166



Init method (*continued*)

- file writer, 190
- generalized operation, 139
- iTool, 69
- visualization, 95

INITIAL\_DATA keyword, 81

initializing superclasses, 69, 96, 127, 140, 167, 191

integer properties, 51

INTEGER property data type, 51

Intelligent Tool *See* iTool

intersection of aggregated properties, 61

IsA method to file reader, 173

iTool

- class, registering, 78
- command line arguments, 80
- component framework *See* framework
- creating, 67
- data object classes, predefined, 38
- data types
  - composite, 34
  - described, 32, 34
  - used by standard iTools, 34
- described
- error handling in launch routine, 82
- help system, 29
- Init method, 69
- instantiating, 83
- keyword arguments, 80
- launch routine, 80
- object class definition file, 69
- object classes
  - documented, 11
  - reference documentation, 10
  - undocumented, 11
- object hierarchy, 21
- simple example, 85
- standard base class, 70
- system object, 21
- system preferences, 64
- user interface architecture, 210

iTool (*continued*)

- user interface object, 212

ITREGISTER, 78, 231

## K

keyword arguments, 80

## L

linestyle properties, 52

LINestyle property data type, 52

location of bitmap resources, 28

## M

messages

- contents, 26
- informational, 221
- observers, 27
- standard, 26
- status, 217

messaging system, 18, 25

## N

NAME property attribute, 59

names, parameter, 41

notification

- described, 25
- message contents, 26
- messages, 18
- observers, 27
- sending, 25
- standard messages, 26
- system, 25

## O

object descriptors, 20

object identifiers

- defined, 19
- described, 18

- object identifiers (*continued*)
  - proxy, 20
- object-oriented programming, 68
- observers, 27, 245
- OnChangeUpdate method, 45, 105
- OnDataDisconnect method, 107
- operations
  - creating, 120
  - data-centric, 125
  - described, 120
  - example, 156
  - IDLitOpBytscl, 122
  - IDLitOpConvolution, 122
  - IDLitOpCurvefitting, 122
  - IDLitOpSmooth, 122
  - pre-defined, 122
  - registering, 72
  - standard base class, 128, 141
  - undo/redo, 123
  - unregistering, 155

## P

- panel widget, 243
- PANEL\_LOCATION keyword, 251
- parameters
  - data types, 32, 41
  - defined, 41
  - names, 41
  - registered, 41
  - registering, 97
- preferences, 48
  - file readers, 64
  - file writers, 64
  - iTool system, 64
  - system, 64
  - visualization, 64
- pre-registered properties, 55
- presentation layer, 18
- ProbeStatusMessage method, 217
- prompts, 219
- PromptUserText method, 220

- PromptUserYesNo method, 219
- properties
  - aggregation, 50, 61, 98
  - attributes, 50, 99
    - defined, 58
    - DESCRIPTION, 59
    - ENUMLIST, 59
    - HIDE, 59
    - NAME, 59
    - PROPERTY\_IDENTIFIER, 59
    - SENSITIVE, 59
    - TYPE, 59
    - UNDEFINED, 59
    - USERDEF, 60
    - VALID\_RANGE, 60
  - data types, 49
    - BOOLEAN, 51
    - COLOR, 51
    - ENUMLIST, 53
    - FLOAT, 51
    - INTEGER, 51
    - LINestyle, 52
    - STRING, 51
    - SYMBOL, 52
    - THICKNESS, 53
    - USERDEF, 51
  - described, 48
  - identifiers, 50, 57
  - interface, 48
  - intersection of aggregated, 61
  - pre-registered, 55
  - registering, 54, 98
  - registration, 50
  - retrieving values, 49
  - setting values, 49
  - sheet, 48
  - union of aggregated, 61
  - update mechanism, 63
  - PROPERTY\_IDENTIFIER property attribute, 59

proxy  
 identifiers, 20  
 registration, 23

## R

RecordFinalValues method, 147  
 RecordInitialValues method, 146  
 RedoOperation method, 150  
 reference documentation for iTool classes, 10  
 REGISTER\_PROPERTIES keyword, 55  
 registered parameter, 41  
 RegisterFileReader method, 177  
 RegisterFileWriter method, 201  
 registering  
 an iTool class, 78  
 callback routines, 248  
 file readers, 72, 177  
 file writers, 72, 201  
 operations, 72, 153  
 parameters, 97  
 properties, 54, 98  
 user interface, services, 231  
 user interface panels, 244, 250  
 visualizations, 71  
 RegisterOperation method, 153  
 RegisterParameter method, 41  
 RegisterProperty method, 54  
 RegisterUIService method, 212, 232  
 RegisterVisualization method, 110  
 RegisterWidget method, 213, 244  
 registration  
 ITREGISTER procedure, 22  
 methods, 22  
 properties, 50  
 proxy, 23  
 Register\* methods, 22  
 visualization types, 110  
 RemoveByIdentifier method, 33  
 REVERSIBLE\_OPERATION property, 123, 154

## S

sending messages, 25  
 sending notifications, 25  
 SENSITIVE property attribute, 59  
 SetData method to file writer, 197  
 SetProperty method  
 and property identifiers, 57  
 data operation, 134  
 file reader, 172  
 file writer, 196  
 generalized operation, 148  
 visualization, 104  
 SetPropertyAttribute method, 58  
 SHOW\_EXECUTION\_UI property, 132, 154  
 status information, providing, 216  
 status messages, 217  
 StatusMessage method, 217  
 string properties, 51  
 STRING property data type, 51  
 superclass initialization, 69, 96, 127, 140, 167, 191  
 symbol properties, 52  
 SYMBOL property data type, 52  
 system object, 21  
 system preferences, 64

## T

thickness properties, 53  
 THICKNESS property data type, 53  
 TYPE  
 property, 250  
 property attribute, 59  
 TYPES property, 154

## U

UI panel *See* user interface panel  
 UI service *See* user interface service  
 UNDEFINED property attribute, 59  
 undo/redo system, 123

- undocumented classes, 11
- UndoExecute method, 135
- UndoOperation method, 149
- union of aggregated properties, 61
- unregistering, 75
  - components, 75
  - file readers, 178
  - file writers, 202
  - generic component, 75
  - operation, 155
  - visualization types, 112
- UnRegisterUIService method, 212
- UnRegisterWidget method, 213
- user defined properties, 51
- user interface
  - architecture, 210
  - elements, 216
  - panel
    - creation routines, 243
    - described, 242
    - example, 252
    - registering, 244, 250
    - TYPE property, 250
  - services
    - creating, 224, 226
    - example, 234
    - executing, 233
    - function, 226
    - IDLitUIHourGlass, 225
    - predefined, 225
    - using, 224
- user interface panels, callback routines, 242
- user interface services, registering, 231
- user interfaces, 18
- USERDEF
  - property attribute, 60
  - property data type, 51

## V

- VALID\_RANGE property attribute, 60
- visualization types
  - creating, 95
  - defined, 90
  - example, 113
  - IDLitVisAxis, 91
  - IDLitVisColorbar, 91
  - IDLitVisContour, 91
  - IDLitVisHistogram, 91
  - IDLitVisImage, 91
  - IDLitVisIsosurface, 92
  - IDLitVisLegend, 92
  - IDLitVisLight, 92
  - IDLitVisPlot, 92
  - IDLitVisPlot3D, 92
  - IDLitVisPolygon, 93
  - IDLitVisPolyline, 93
  - IDLitVisRoi, 93
  - IDLitVisSurface, 93
  - IDLitVisText, 93
  - IDLitVisVolume, 94
  - predefined, 91
  - preferences, 64
  - registering, 71, 110
  - standard base class, 97
  - unregistering, 112
- VISUALIZATION\_TYPE keyword, 84

## W

- widgets, 210