IDL

External Development Guide

RSI
Research Systems Inc.

IDL Version 6.0
July, 2003 Edition

Copyright © Research Systems, Inc.
All Rights Reserved

0703IDL60EDG

Restricted Rights Notice

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

Numerical Recipes™ is a trademark of Numerical Recipes Software. Numerical Recipes routines are used by permission.

GRG2™ 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.

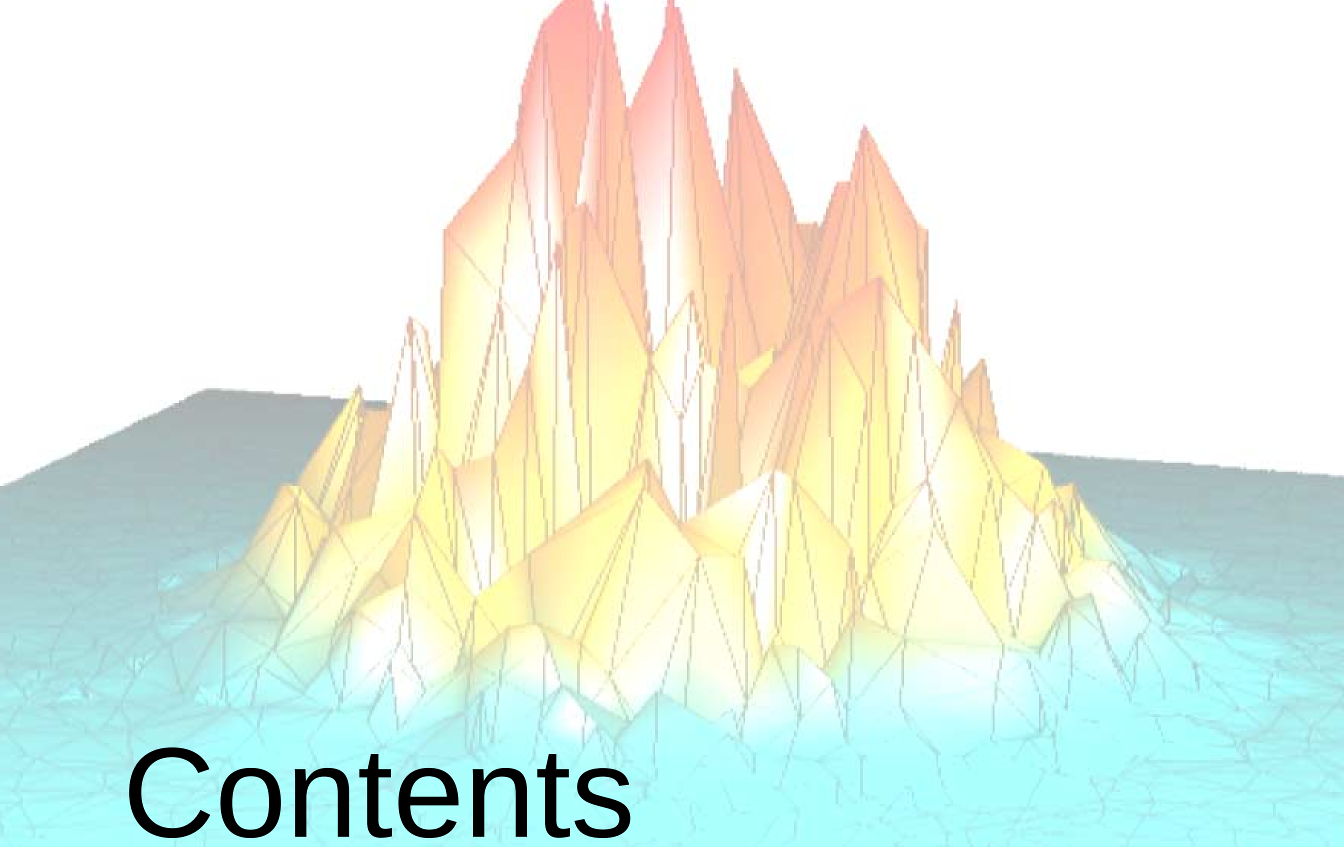
Use of this software for providing LZW capability for any purpose is not authorized unless user first enters into a license agreement with Unisys under U.S. Patent No. 4,558,302 and foreign counterparts. For information concerning licensing, please contact: Unisys Corporation, Welch Licensing Department - 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.

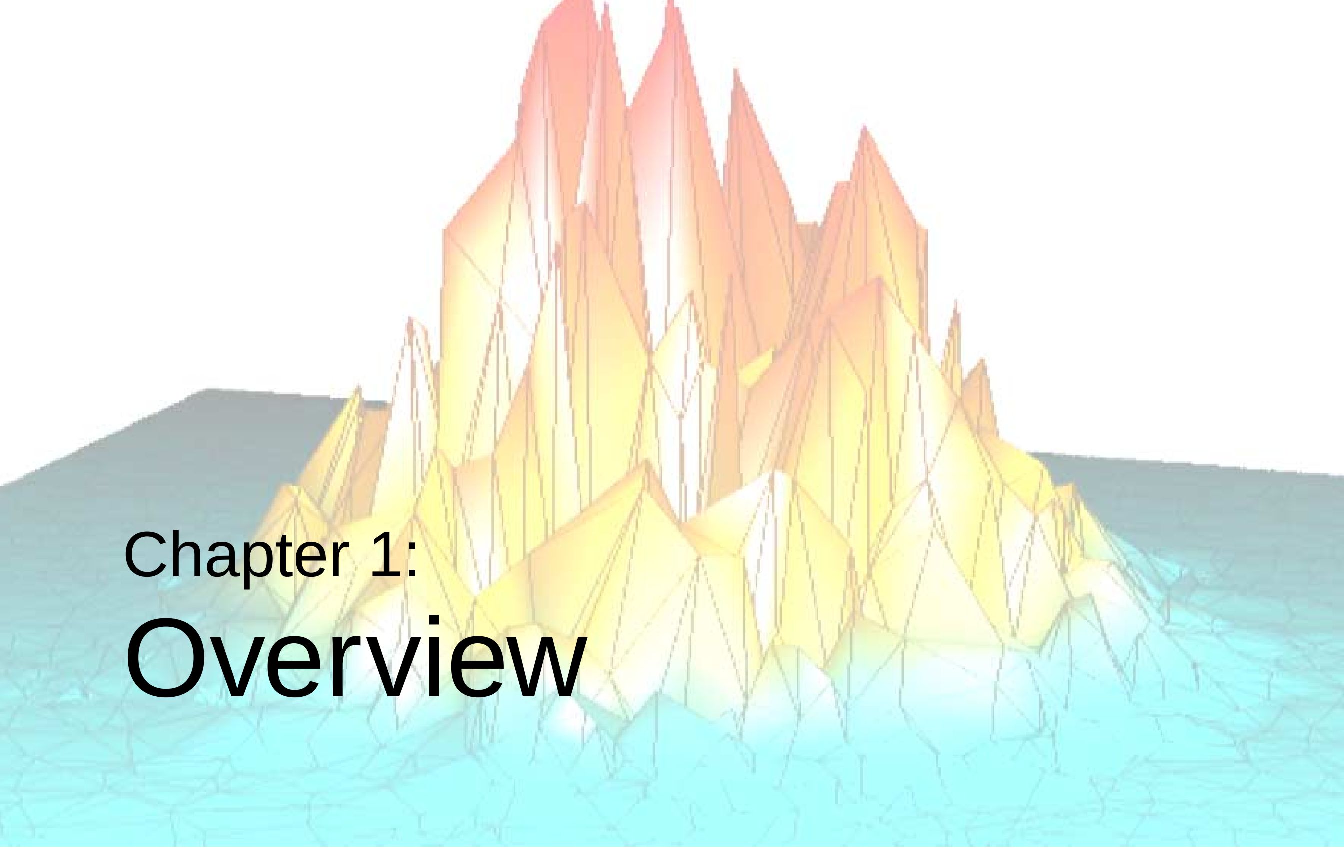
This product includes software developed by the Apache Software Foundation (http://www.apache.org/)

IDL Wavelet Toolkit Copyright © 2002 Christopher Torrence.

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



Contents



Chapter 1: Overview

This chapter discusses the following topics:

About this Manual

The *External Development Guide* describes options for using code not written in the IDL language alongside IDL itself. It is divided into three parts:

Part I: Techniques That Do Not Use IDL's Internal API

This section discusses techniques that allow IDL to work together with programs written in other programming languages, using IDL's "public" interfaces. Little or no familiarity with IDL's internal interfaces is required. For many users, the techniques in this section will solve most problems that require IDL to use — or be used by — other programs. Topics covered in Part I include:

- Letting IDL programs interact with UNIX programs via pipes.
- Incorporating COM objects and ActiveX controls into IDL programs.
- Giving Microsoft Windows programs access to IDL features via the IDLDrawWidget ActiveX control.
- Incorporating Java objects into IDL programs.
- Using IDL as a Remote Procedure Call server on a UNIX system.
- Calling routines written in other programming languages from within IDL using the CALL_EXTERNAL function.

Part II: IDL's Internal API

This section describes IDL's internal implementation in enough detail to allow you to write an IDL system routine in another compiled programming language (usually C) and link it with IDL.

Part III: Techniques That Use IDL's Internal API

This section describes the process of combining IDL with code written in another programming language. Topics covered in Part III include:

- Creating a system routine using the interface described in Part II and linking that routine into IDL at runtime.
- Calling IDL as a subroutine from another program ("Callable IDL").
- Adding user-defined widgets to IDL widget applications.

Supported Inter-Language Communication Techniques in IDL

IDL supports a number of different techniques for communicating with the operating system and programs written in other languages. These methods are described, in brief, below.

Options are presented in approximate order of increasing complexity. We recommend that you favor the simpler options at the head of this list over the more complex ones that follow if they are capable of solving your problem.

It can be difficult to choose the best option — there is a certain amount of overlap between their abilities. We highlight the advantages and disadvantages of each method as well as make recommendations to help you decide which approach to take. By comparing this list with the requirements of the problem you are trying to solve, you should be able to quickly determine the best solution.

Translate into IDL

Advantages

All the benefits of using a high level, interpreted, array oriented environment with high levels of platform independence.

Disadvantages

Not always possible.

Recommendation

Writing in IDL is the easiest path. If you have existing code in another language that is simple enough to translate to IDL, this is the best way to go. You should investigate the other options if the existing code is sufficiently complex, has desirable performance advantages, or is the reference implementation of some standardized package. Another good reason for considering the techniques described in this book is if you wish to access IDL abilities from a large program written in some other language.

SPAWN

The simplest (but most limited) way to access programs external to IDL is to use the SPAWN procedure. Calling SPAWN spawns a child process that executes a specified command. The output from SPAWN can be captured in an IDL string variable. Under UNIX, IDL can communicate with a child process through a bi-directional pipe using SPAWN. More information about SPAWN can be found in Chapter 2, "Using SPAWN and UNIX Pipes" or in the documentation for "SPAWN" in the *IDL Reference Guide* manual.

Advantages

- Simplicity
- Allows use of existing standalone programs.
- Under UNIX, data can be sent to and returned by the program via a pipe, making sophisticated inter-program communication possible quickly and easily.

Disadvantages

- Non-UNIX hosts are unable to use the pipe facility to communicate with the program. Data can only be sent to the command via arguments to SPAWN.

Recommendation

SPAWN is the easiest form of interprocess communication supported by IDL and allows accessing operating system commands directly.

Microsoft COM and ActiveX

IDL supports the inclusion of COM objects and ActiveX controls within IDL applications running on Microsoft Windows systems by encapsulating the object or control in an IDL object. Full access to the COM object or ActiveX control's methods is available in this manner, allowing you to incorporate features not available in IDL into IDL programs. For more information, see Chapter 3, "Overview: COM and ActiveX in IDL".

IDL also provides the IDLDrawWidget ActiveX control. The IDLDrawWidget control is built around IDL for Windows and provides an easy mechanism for integrating IDL with Microsoft Windows applications written in languages such as C, C++, Visual Basic, Fortran, Delphi, and others. For more information, see Chapter 6, "The IDLDrawWidget ActiveX Control".

Advantages

- Integrates easily with an important interprocess communication mechanism under Microsoft Windows.
- May support a higher level interface than the function call interfaces supported by the remaining options.

Disadvantages

- Only supported under Microsoft Windows.

Recommendation

Incorporate COM objects or ActiveX controls into your Windows-only IDL application if doing so provides functionality you cannot easily duplicate in IDL.

Use the IDL ActiveX control if you are writing a Windows-only application in a language that supports ActiveX and you wish to use IDL to perform computation or graphics within a framework established by this other application.

Sun Java

IDL also supports the inclusion of Java objects within IDL applications by encapsulating the object or control in an IDL object. Full access to the Java object is available in this manner, allowing you to incorporate features not available in IDL into IDL programs. For more information, see Chapter 8, “Using Java Objects in IDL”.

Advantages

- Integrates easily with all types of Java code.
- Can easily leverage existing Java objects into IDL.

Disadvantages

- Only supported under Microsoft Windows, Linux, Solaris, and Macintosh platforms supported in IDL.

Recommendation

Incorporate Java objects into your IDL application if doing so provides functionality you cannot easily duplicate in IDL.

UNIX Remote Procedure Calls (RPCs)

UNIX platforms can use Remote Procedure Calls (RPCs) to facilitate communication between IDL and other programs. IDL is run as an RPC server and your own program is run as a client. IDL's RPC functionality is documented in Chapter 10, "Remote Procedure Calls".

Advantages

- Code executes in a process other than the one running IDL, possibly on another machine, providing robustness and protection in a distributed framework.
- API is similar to that employed by Callable IDL, making it reasonable to switch from one to the other.
- Possibility of overlapped execution on a multi-processor system.

Disadvantages

- Complexity of managing RPC servers.
- Bandwidth limitations of network for moving large amounts of data.
- Only supported under UNIX.

Recommendation

Use RPC if you are coding in a distributed UNIX-only environment and the amount of data being moved is reasonable on your network. CALL_EXTERNAL might be more appropriate for especially simple tasks, or if the external code is not easily converted into an RPC server, or you lack RPC experience and knowledge.

CALL_EXTERNAL

IDL's CALL_EXTERNAL function loads and calls routines contained in shareable object libraries. IDL and the called routine share the same memory and data space. CALL_EXTERNAL is much easier to use than either system routines (LINKIMAGE, DLMs) or Callable IDL and is often the best (and simplest) way to communicate with other programs. CALL_EXTERNAL is also supported on all IDL platforms.

While many of the topics in this book can enhance your understanding of CALL_EXTERNAL, specific documentation and examples can be found in Chapter 9, “CALL_EXTERNAL” and the documentation for “CALL_EXTERNAL” in the *IDL Reference Guide* manual.

Advantages

- Allows calling arbitrary code written in other languages.
- Requires little or no understanding of IDL internals.

Disadvantages

- Errors in coding can easily corrupt the IDL program.
- Requires understanding of system programming, compiler, and linker.
- Data must be passed to and from IDL in precisely the correct type and size or memory corruption and program errors will result.
- System and hardware dependent, requiring different binaries for each target system.

Recommendation

Use CALL_EXTERNAL to call code written for general use in another language (that is, without knowledge of IDL internals). For safety, you should call your CALL_EXTERNAL functions within special IDL procedures or functions that do error checking of the inputs and return values. In this way, you can reduce the risks of corruption and give your callers an appropriate IDL-like interface to the new functionality. If you use this method to incorporate external code into IDL, RSI highly recommends that you also use the MAKE_DLL procedure and the AUTO_GLUE keyword to CALL_EXTERNAL.

If you lack knowledge of IDL internals, CALL_EXTERNAL is the best way to add external code quickly. Programmers who do understand IDL internals will often write a system routine instead to gain flexibility and full integration into IDL.

IDL System Routine (LINKIMAGE, DLMs)

It is possible to write system routines for IDL using a compiled language such as C. Such routines are written to have the standard IDL calling interface, and are dynamically linked, as with CALL_EXTERNAL. They are more difficult to write, but more flexible and powerful. System routines provide access to variables and other objects inside of IDL.

This book contains the information necessary to successfully add your own code to IDL as a system routine. Especially important is Chapter 21, “Adding System Routines”. Additional information about system routines can be found in Chapter 9, “CALL_EXTERNAL” and in the documentation for “LINKIMAGE” in the *IDL Reference Guide* manual.

Advantages

- This is the most fully integrated option. It allows you to write IDL system routines that are indistinguishable from those written by RSI.
- In use, system routines are very robust and fault tolerant.
- Allows direct access to IDL user variables and other important data structures.

Disadvantages

- All the disadvantages of CALL_EXTERNAL.
- Requires in-depth understanding of IDL internals, discussed in Part II of this manual.

Recommendation

Use system routines if you require the highest level of integration of your code into the IDL system. UNIX users with RPC experience should consider using RPCs to get the benefits of distributed processing. If your task is sufficiently simple or you do not have the desire or time to learn IDL internals, CALL_EXTERNAL is an efficient way to get the job done.

Callable IDL

IDL is packaged in a shareable form that allows other programs to call IDL as a subroutine. This shareable portion of IDL can be linked into your own programs. This use of IDL is referred to as “Callable IDL” to distinguish it from the more usual case of calling your code *from* IDL via CALL_EXTERNAL or as a system routine (LINKIMAGE, DLM).

This book contains the information necessary to successfully call IDL from your own code.

Advantages

- Supported on all systems.
- Allows extremely low level access to IDL.

Disadvantages

- All the disadvantages of CALL_EXTERNAL or IDL system routines.
- IDL imposes some limitations on programming techniques that your program can use.

Recommendation

Most platforms offer a specialized method to call other programs that might be more appropriate. Windows users should consider the ActiveX control or COM component. UNIX users should consider using the IDL RPC server. If these options are not appropriate for your task and you wish to call IDL from another program, then use Callable IDL.

Dynamic Linking Terminology and Concepts

All systems on which IDL runs support the concept of dynamic linking. Dynamic linking consists of compiling and linking code into a form which is loadable by programs at run time as well as link time. The ability to load them at run time is what distinguishes them from ordinary object files. Various operating systems have different names for such loadable code:

- UNIX: Sharable Libraries
- Windows: Dynamic Link Libraries (DLL)

In this manual, we will call such files *sharable libraries* in order to have a consistent and uniform way to refer to them. It should be understood that this is a generic usage that applies equally to all of these systems. Sharable libraries contain functions that can be called by any program that loads them. Often, you must specify special compiler and linker options to build a sharable library. On many systems, the linker gives you control over which functions and data (often referred to as *symbols*) are visible from the outside (public symbols) and which are hidden (private symbols). Such control over the interface presented by a sharable library can be very useful. Your system documentation discusses these options and explains how to build a sharable library.

Dynamic linking is the enabling technology for many of the techniques discussed in this manual. If you intend to use any of these techniques, you should first be sure to study your system documentation on this topic.

CALL_EXTERNAL

CALL_EXTERNAL uses dynamic linking to call functions written in other languages from IDL.

LINKIMAGE and Dynamically Loadable Modules (DLMs)

These mechanisms use dynamic linking to add external code that supports the standard IDL system routine interface to IDL as system routines.

Callable IDL

Most of IDL is built as a sharable library. The actual IDL program that implements the standard interactive IDL program links to this library and uses it to do its work. Since IDL is a sharable library, it can be called by other programs.

Remote Procedure Calls (RPCs)

The IDL RPC server is a program that links to the IDL sharable library. The IDL RPC client side library is also a sharable library. Your RPC client program links against it to obtain access to the IDL RPC system.

When is it Appropriate to Combine External Code with IDL?

IDL is an interactive program that runs across numerous operating systems and hardware platforms. The IDL user enjoys a large amount of portability across these platforms because IDL provides access to system abilities at a relatively high level of abstraction. The large majority of IDL users have no need to understand its inner workings or to link their own code into it.

There are, however, reasons to combine external code with IDL:

- Many sites have an existing investment in other code that they would prefer to use from IDL rather than incurring the cost of rewriting it in the IDL language.
- It is often best to use the reference implementation of a software package rather than re-implement it in another language, risk adding incorrect behaviors to it, and incur the ongoing maintenance costs of supporting it.
- IDL may be largely suitable for a given task, requiring only the addition of an operation that cannot be performed efficiently in the IDL language.

A programmer who is considering adding compiled code to IDL should understand the following caveats:

- RSI attempts to keep the interfaces described in this document stable, and we endeavor to minimize gratuitous change. However, we reserve the right to make any changes required by the future evolution of the system. Code linked with IDL is more likely to require updates and changes to work with new releases of IDL than programs written in the IDL language.
- The act of linking compiled code to IDL is inherently less portable than use of IDL at the user level.
- Troubleshooting and debugging such applications can be very difficult. With standard IDL, malfunctions in the program are clearly the fault of RSI, and given a reproducible bug report, we attempt to fix them promptly. A program that combines IDL with other code makes it difficult to unambiguously determine where the problem lies. The level of support RSI can provide in such troubleshooting is minimal. The programmer is responsible for locating the source of the difficulty. If the problem is in IDL, a simple program demonstrating the problem must be provided before we can address the issue.

Skills Required to Combine External Code with IDL

There is a large difference between the level at which a typical user sees IDL compared to that of the internals programmer. To the user, IDL is an easy-to-use, array-oriented language that combines numerical and graphical abilities, and runs on many platforms. Internally, IDL is a large C language program that includes a compiler, an interpreter, graphics, mathematical computation, user interface, and a large amount of operating system-dependent code.

The amount of knowledge required to effectively write internals code for IDL can come as a surprise to the user who is only familiar with IDL's external face. To be successful, the programmer must have experience and proficiency in many of the following areas:

Microsoft COM

To incorporate a COM object into your IDL program, you should be familiar with COM interfaces in general and the interface of the object you are using in particular.

Microsoft ActiveX

To incorporate an ActiveX control into your IDL widget application, you should be familiar with COM interfaces in general and the interface of the control you are using in particular.

To use the IDLDrawWidget ActiveX control, you should be familiar with the programming environment in which you will be using the control (Visual Basic, for example). A level of understanding of ActiveX and COM is necessary.

Sun Java

To incorporate a Java object into your IDL program, you should be familiar with Java object classes in general and the methods and data members of the object you are using in particular.

UNIX RPC

To use IDL as an RPC server, a knowledge of Sun RPC (Also known as ONC RPC) is required. Sun RPC is the fundamental enabling technology that underlies the popular NFS (Network File System) software available on all UNIX systems, and as such, is universally available on UNIX. The system documentation on this subject should be sufficient.

ANSI C

IDL is written in ANSI C. To understand the data structures and routines described in this document, you must have a complete understanding of this language.

System C Compiler, Linker, and Libraries

In order to successfully integrate IDL with your code, you must fully understand the compilation tools being used as well as those used to build IDL and how they might interact. IDL is built with the standard C compiler used (and usually supplied) by the vendor of each platform to ensure full compatibility with all system components.

Inter-language Calling Conventions (C++, Fortran, …)

It is possible to link IDL directly with code written in compiled languages other than C although the details differ depending on the machine, language, and compiler used. It is the programmer's responsibility to understand the inter-language calling conventions and rules for the target environment—there are too many possibilities for RSI to actively document them all. ANSI C is a standard system programming language on all systems supported by IDL, so it is usually straightforward to combine it with code written in other compiled languages. You need to understand:

- The conventions used to pass parameters to functions in both languages. For example, C uses call-by-value while Fortran uses call-by-reference. It is easy to compensate for such conventions, but they must be taken into account.
- Any systematic name changes applied by the compilers. For example, some compilers add underscores at the beginning or end of names of functions and global data.
- Any run-time initialization that must be performed. On many systems, the real initial entry point for the program is not main(), but a different function that performs some initialization work and then calls your main() function. Usually these issues have been addressed by the system vendor, who has a large interest in allowing such inter-language usage:
    - If you call IDL from a program written in a language other than C, has the necessary initialization occurred?
    - If you use IDL to call code written in a language other than C, do you need to take steps to initialize the runtime system for that language?
    - Are the two runtime systems compatible?

Alternatives to direct linking (Microsoft COM or Active X) exist on some systems that simplify the details of inter-language linking.

C++

We are often asked if IDL can call C++ code. Compatibility with C has always been a strong design goal for C++, and C++ is largely a superset of the C language. It certainly is possible to combine IDL with C++ code. Callable IDL is especially simple, as all you need to do is to include the idl_export.h header file in your C++ code and then call the necessary IDL functions directly. Calling C++ code from IDL (CALL_EXTERNAL, System Routines) is also possible, but there are some issues you should be aware of:

- As a C program, IDL is not able to directly call C++ methods, or use other object-oriented features of the C++ language. To use these C++ features, you must supply a function with C linkage (using an extern “C” specification) for IDL to call. That routine, which is written in C++ is then able to use the C++ features.
- IDL does not initialize any necessary C++ runtime code. Your system may require such code to be executed before your C++ code can run. Consult your system documentation for details. (Please be aware that this information can be difficult to find; locating it may require some detective work on your part.)

Fortran

Issues to be aware of when combining IDL with Fortran:

- The primary issue surrounding the calling of Fortran code from IDL is one of understanding the calling conventions of the two languages. C passes everything by value, and supplies an operator that lets you explicitly take the address of a memory object. Fortran passes everything by reference (by address). Difficulties in calling FORTRAN from C usually come down to handling this issue correctly. Some people find it helpful to write a C wrapper function to call their Fortran code, and then have IDL call the wrapper. This is generally not necessary, but may be convenient.
- IDL is a C program, and as such, does not initialize any necessary Fortran runtime code. Your system may require such code to be executed before your Fortran code can run. In particular, Fortran code that does its own input output often requires such startup code to be executed. Consult your system documentation for details. One common strategy that can minimize this sort of problem is to use IDL’s I/O facilities to do I/O, and have your Fortran code limit itself to computation.

Operating System Features And Conventions

With the exception of purely numerical code, the programmer must usually fully understand the target operating system environment in which IDL is running in order to write code to link with it.

Microsoft Windows

You must be an experienced Windows programmer with an understanding of 32–bit applications, WIN32, and DLLs.

UNIX

You should understand system calls, signals, processes, standard C libraries, and possibly even X Windows depending on the scope of the code being linked.

IDL Organization

In order to properly write code to be linked with IDL, it is necessary to understand a little about its internal operation. This section is intended to give just enough background to understand the material that follows. Traditional interpreted languages work according to the following algorithm:

```
while (statements remaining) {
  Get next statement.
  Perform lexical analysis and parse statement.
  Execute statement.
}
```

This description is accurate at a conceptual level, and most early interpreters did their work in exactly this way due to its simplicity. However, this scheme is inefficient because:

- The meaning of each statement is determined by the relatively expensive operations of lexical analysis, parsing, and semantic analysis each and every time the statement is encountered.
- Since each statement is considered in isolation, any statement that requires jumping to a different location in the program will require an expensive search for the target location. Usually, this search starts at the top of the file and moves forward until the target is found.

To avoid these problems, the IDL system uses a two-step process in which compilation and interpretation are separate. The core of the system is the interpreter. The interpreter implements a simple, stack-based postfix language, in which each instruction corresponds to a primitive of the IDL language. This internal form is a compact binary version of the IDL language routine. Routines written in the IDL language are compiled into this internal form by the IDL compiler when the .RUN executive command is issued, or when any other command requires a new routine to be executed. Once the IDL routine is compiled, the original version is ignored, and all references to the routine are to the compiled version. Some of the advantages of this organization are:

- The expensive compilation process is only performed once, no matter how often the resulting code is executed.
- Statements are not considered in isolation, so the compiler keeps track of the information required to make jumping to a new location in the program fast.

- The binary internal form is much faster to interpret than the original form.
- The internal form is compact, leading to better use of main memory, and allowing more code to fit in any memory cache the computer might be using.

The Interpreter Stack

The primary data structure in the interpreter is the stack. The stack contains pointers to variables, which are implemented by **IDL_VARIABLE** structures (see "The IDL_VARIABLE Structure" on page 267). Pointers to **IDL_VARIABLEs** are referred to as **IDL_VPTR**s. Most interpreter instructions work by removing a predefined number of elements from the stack, performing their function, and then pushing the **IDL_VPTR** to the resulting **IDL_VARIABLE** back onto the stack. The removed items are the arguments to the instruction, and the new element represents the result. In this sense, the IDL interpreter is no different from any other postfix language interpreter. When an IDL routine is compiled, the compiler checks the number of arguments passed to each system routine against the minimum and maximum number specified in an internal table of routines, and signals an error if an invalid number of arguments is specified.

At execution time, the interpreter instructions that execute system procedures and functions operate as follows:

1. Look up the requested routine in the internal table of routines.
2. Execute the routine that implements the desired routine.
3. Remove the arguments from the stack.
4. If the routine was a function, push its result onto the stack.

Thus, the compiler checks for the proper number of arguments, and the interpreter does all the work related to pushing and popping elements from the stack. The called function need only worry about executing its operation and providing a result.

External Definitions

The file `idl_export.h`, found in the `external/include` subdirectory of the IDL distribution, supplies all the IDL-specific definitions required to write code for inclusion with IDL. As such, this file defines the interface between IDL and your code. It will be worth your while to examine this file, reading the comments and getting a general idea of what is available. If you are not writing in C, you will have to translate the definitions in this file to suit the language you are using.

Warning

`idl_export.h` contains some declarations which are necessary to the compilation process, but which are still considered private to RSI. Such declarations are likely to be changed in the future and should not be depended on. In particular, many of the structure data types discussed in this document have more fields than are discussed here—such fields should not be used. For this reason, you should always include `idl_export.h` rather than entering the type definitions from this document. This will also protect you from changes to these data structures in future releases of IDL. Anything in `idl_export.h` that is not explicitly discussed in this document should not be relied upon.

The following two lines should be included near the top of every C program file that is to become part of IDL:

```
#include <stdio.h>
#include "idl_export.h"
```

Interpreting Logical Boolean Values

IDL is written in the C programming language, and this manual therefore discusses C language functions and data structures from the IDL program. In this documentation, you will see references to logical (boolean) arguments and results referred to in any of the following forms: True, False, TRUE, FALSE, IDL_TRUE, IDL_FALSE, and possibly other permutations on these. In all cases, the meaning of true and false in this manual correspond to those of the C programming language: A zero (0) value is interpreted as "false", and a non-zero value is "true".

When reading this manual, please be aware of the following points:

- Unless otherwise specified, the actual word used when discussing logical values is not important (i.e. true, True, TRUE, and IDL_TRUE) all mean the same thing.
- Internally, IDL uses the IDL_TRUE and IDL_FALSE macros described in "Macros" on page 409, for hard-wired logical constants. These macros have the values 1, and 0 respectively. This convention is nothing more than reflection of the need for a consistent standard within our code, and a desire to keep IDL names within a standard namespace to avoid collisions with user selected names. Otherwise, any of those other alternative names might have been used with equally good results.
- We don't use the IDL_TRUE and IDL_FALSE convention in the text of this book because it would be unnecessarily awkward, preferring the more natural True/TRUE and False/FALSE.
- The convention for truth values in the IDL Language differ from those used in the C language. It is important to keep the language being used in mind when reading code to avoid drawing incorrect conclusions about its meaning.

Compilation And Linking Details

Once you've written your code, you need to compile it and link it into IDL before it can be run. Information on how to do this is available in the various subdirectories of the `external` subdirectory of the IDL distribution. References to files that are useful in specific situations are contained in this book.

In addition:

- The IDL MAKE_DLL procedure, documented in the *IDL Reference Manual*, provides a portable high level mechanism for building sharable libraries from code written in the C programming language.
- The IDL !MAKE_DLL system variable is used by the MAKE_DLL procedure to construct C compiler and linker commands appropriate for the target platform. If you do not use MAKE_DLL to compile and link your code, you may find the value of !MAKE_DLL.CC and !MAKE_DLL.LD helpful in determining which options to specify to your compiler and linker, in conjunction with your system and compiler documentation. For the C language, the options in !MAKE_DLL should be very close to what you need. For other languages, the !MAKE_DLL options should still be helpful in determining which options to use, as on most systems, all the language compilers accept similar options.
- The UNIX IDL distribution has a `bin` subdirectory that contains platform specific directories that in turn hold the actual IDL binary and related files. Included with these files is a `Makefile` that shows how to build IDL from the shareable libraries present in the directory. The link line in this makefile should be used as a starting point when linking your code with Callable IDL—simply omit `main.o` and include your own object files, containing your own main program.
- A more detailed description of the issues involved in compiling and linking your code can be found in this book under "Compiling Programs That Call IDL" on page 476.

Recommended Reading

There are many books written on the topics discussed in the previous section. The following list includes books we have found to be the most useful over the years in the development and maintenance of IDL. There are thousands of books not mentioned here. Some of them are also excellent. The absence of a book from this list should not be taken as a negative recommendation.

The C Language

Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language, Second Edition.* Englewood Cliffs, New Jersey: Prentice Hall, 1988. ISBN 0-13-110370-9. This is the original C language reference, and is essential reading for this subject.

In addition, you should study the vendor supplied documentation for your compiler.

Microsoft Windows

The following books will be useful to anyone building IDL system routines or applications that call IDL in the Microsoft Windows environment.

Petzold, Charles. *Programming Windows, The Definitive Guide to the Win32 API*, Microsoft Press, 1998. ISBN 157231995X (Supersedes: *Programming Windows 95*).

Richter, Jeffrey. *Programming Applications for Microsoft Windows*. Microsoft Press, 1999. ISBN 1572319968 (Supersedes: Advanced Windows, Third Edition).

The Microsoft Developer Network (MSDN) supplies essential documentation for programming in the Windows environment. This documentation is part of the Visual C++ environment. More information on the MSDN is available at `http://msdn.microsoft.com`.

Sun Java

Flanagan, David. *Java in a Nutshell, Fourth Edition*, O'Reilly & Associates, March 2002. ISBN 0596002831. This book provides an accelerated introduction to the Java language and key APIs.

In addition, you should study the Java tutorials and documentation provided on the Sun's Java website (http://www.java.sun.com).

UNIX

Stevens, W. Richard. *Advanced Programming in the UNIX Environment.* Reading, Massachusetts: Addison Wesley, 1992. ISBN 0-201-56317-7. This is the definitive reference for UNIX system programmers. It covers all the important UNIX concepts and covers the major UNIX variants in complete detail.

Rochkind, Marc J. *Advanced UNIX Programming.* Englewood Cliffs, New Jersey: Prentice Hall, 1985. ISBN 0-13-011818-4. This volume is also extremely well written and does an excellent job of explaining and motivating the fundamental UNIX concepts that underlie the UNIX system calls. This book suffers in comparison to the Stevens book in that it discusses older UNIX systems rather than current systems and lacks discussion of networking. However, what it does cover is correct and very readable, and it is much shorter than Stevens.

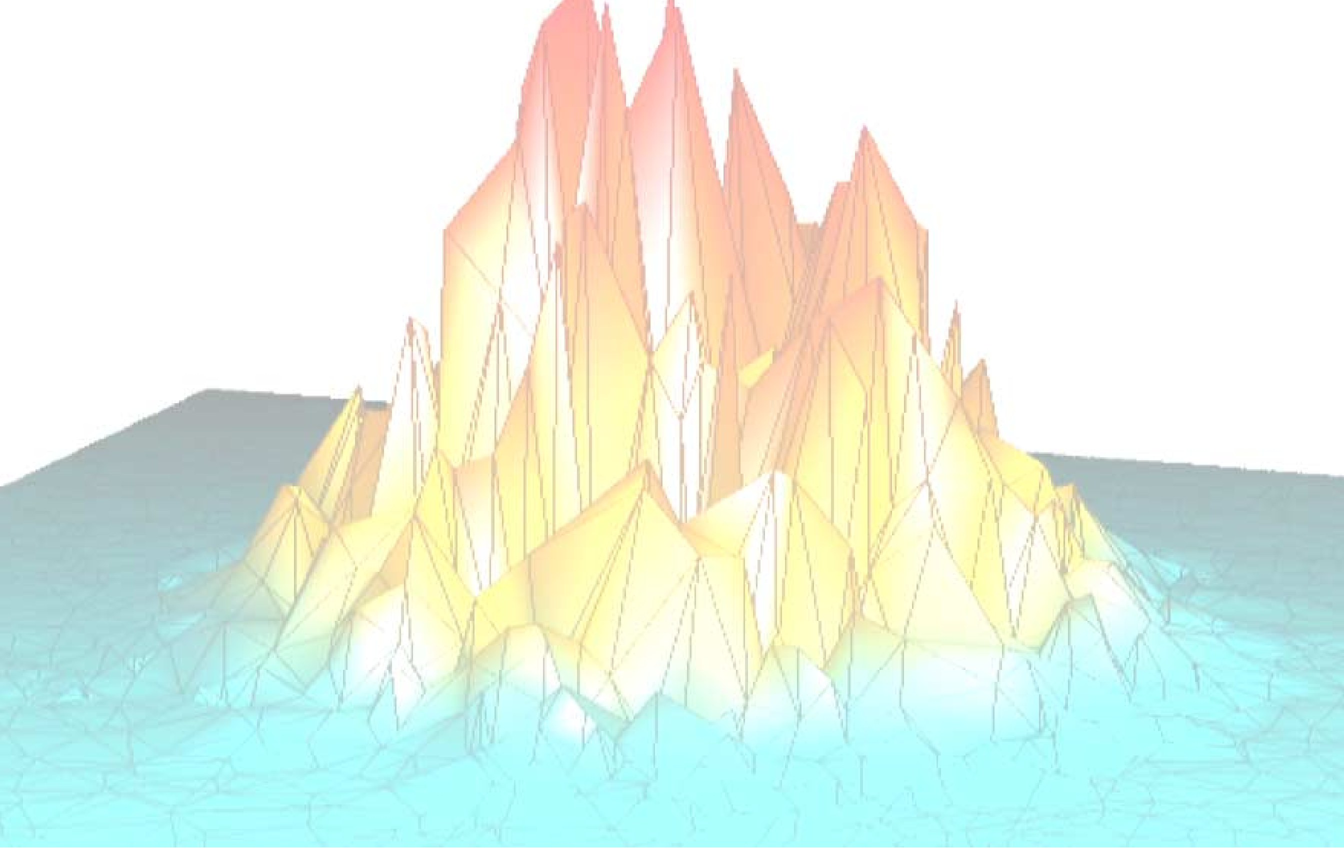
The vendor-supplied documentation and manual pages should be used in combination with the books listed above.

X Windows

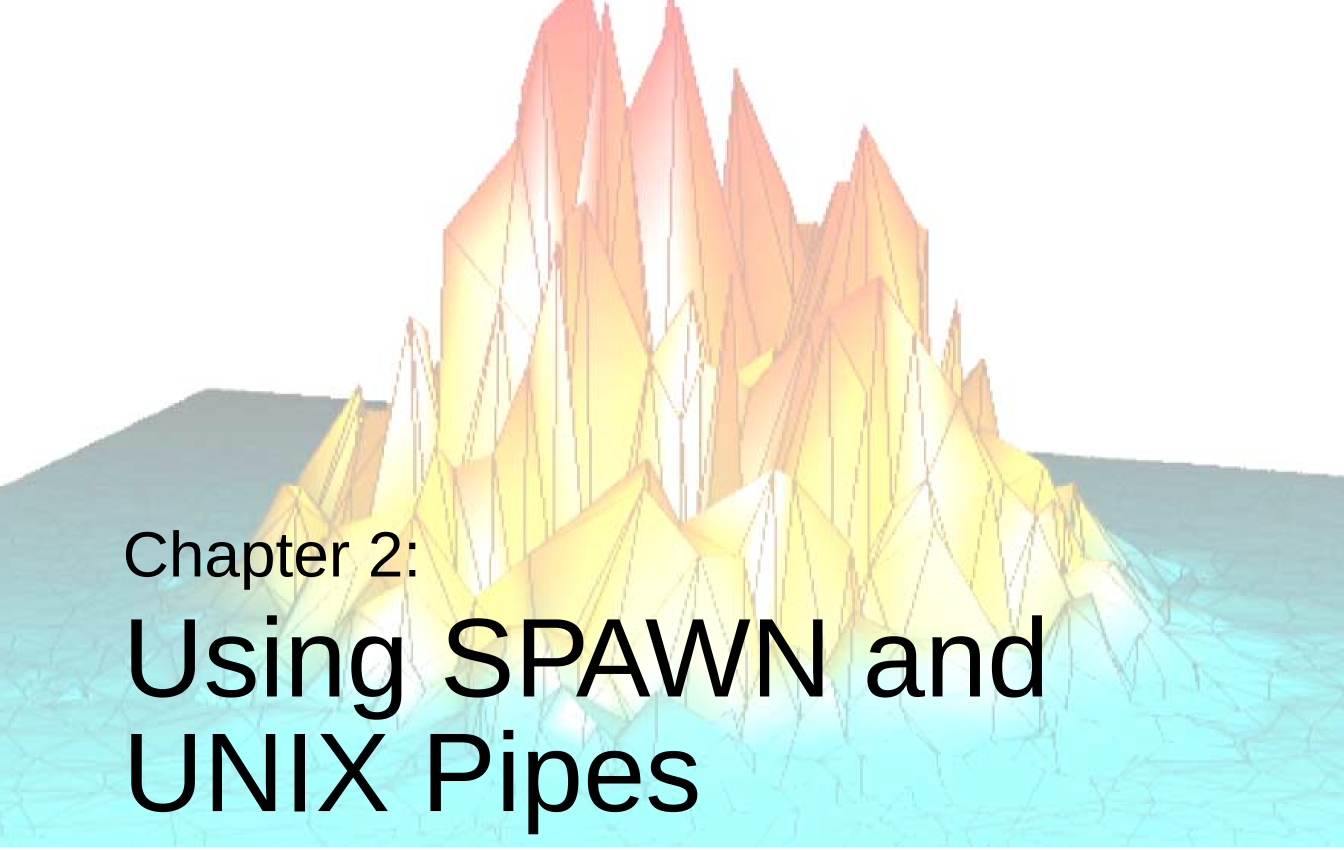
The X Windows series by O’Reilly & Associates contains all the information needed to program for the X Window system. There are several volumes—the ones you will need depend on the type of programming you are doing.

Scheifler, Robert W. and James Gettys. *X Window System.* Digital Press. This is purely a reference manual, as opposed to the O’Reilly books which contain a large amount of tutorial as well as reference information. This book is primarily useful for those using XLIB to draw graphics into Motif Draw Widgets and for those who need to understand the base layers of X Windows. Motif programmers may not require this information since Motif hides many of these details.

There are many other X Windows books on the market with varying levels of quality and usefulness. Note that most X Windows books are updated with each version of the system. (X Version 11, Release 6 is the current version at this printing.)



Part I: Techniques That Do Not Use IDL’s Internal API



Chapter 2: Using SPAWN and UNIX Pipes

IDL’s SPAWN procedure spawns a child process to execute a command or series of commands. Cross-platform use of SPAWN is described in detail in the *IDL Reference Guide*. This section describes a procedure available only on UNIX systems: communicating with the spawned child process using UNIX pipes.

By default, calls to the SPAWN procedure cause the IDL process to wait until the child process has finished before continuing. On UNIX systems, IDL can attach a bidirectional pipe to the standard input and output of the child process, and then continue without waiting for the child process to finish. The pipe created in this manner appears in the IDL process as a normal logical file unit.

Once a process has been started in this way, the normal IDL input/output facilities can be used to communicate with it. The ability to use a child process in this manner allows you to solve specialized problems using other languages and to take advantage of existing programs.

In order to start such a process, use the UNIT keyword to SPAWN to specify a named variable in which the logical file unit number will be stored. Once the child process has done its work, use the FREE_LUN procedure to close the pipe and delete the process.

When using a child process in this manner, it is important to understand the following points:

- Closing the file unit causes the child process to be killed. Therefore, do not close the unit until the child process completes its work.
- A UNIX pipe is simply a buffer maintained by the operating system. It has a fixed length and can therefore become completely filled. When this happens, the operating system puts the process that is filling the pipe to sleep until the process at the other end consumes the buffered data. The use of a bidirectional pipe can lead to deadlock situations in which both processes are waiting for the other. This can happen if the parent and child processes do not synchronize their reading and writing activities.
- Most C programs use the input/output facilities provided by the Standard C Library (*stdio*). In situations where IDL and the child process are carrying on a running dialog (as opposed to a single transaction), the normal buffering performed by *stdio* on the output file can cause communications to hang. We recommend calling the *stdio setbuf()* function as the first statement of the child program to eliminate such buffering.

  ```
  (void) setbuf(stdout, (char *) 0);
  ```

  It is important that this statement occur before any output operation is executed; otherwise, it may not have any effect.

Example: Communicating with a Child Process Under UNIX

The C program shown in the following example (`test_pipe.c`) accepts floating-point values from its standard input and returns their average on the standard output. In actual practice, such a trivial program would never be used from IDL, since it is simpler and more efficient to perform the calculation within IDL itself. The example does, however, serve to illustrate a method by which significant programs can be called from IDL.

In the interest of brevity, some error checking that would normally be included in such a program has been omitted. For example, a real program would need to check the non-zero return values from `fread(3)` and `fwrite(3)` to ensure that the desired amount of data was actually transferred.

This program performs the following steps:

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

main()
{
  float *data, total = 0.0;
  char *err_str;
  int i, n;

  /* Make sure the output is not buffered */
  setbuf(stdout, (char *) 0);

  /* Find out how many points */
  if (!fread(&n, sizeof(n), 1, stdin)) goto error;

  /* Get memory for the array */
  if (!(data = (float *) malloc(n * sizeof(*data)))) goto error;

  /* Read the data */
  if (!fread(data, sizeof(*data), n, stdin)) goto error;

  /* Calculate the average */
  for (i=0; i < n; i++) total += data[i];
  total /= (float) n;

  /* Return the answer */
  if (!fwrite(&total, sizeof(*data), 1, stdout)) goto error;
  return;

error:
  err_str = strerror(errno);
  if (!err_str) err_str = "<unknown error>";
  fprintf(stderr, "test_pipe: %s\n", err_str);
}
```

Table 2-1: test_pipe.c

1. Reads a long integer that tells how many data points to expect, because it is desirable to be able to average an arbitrary number of points.
2. Obtains dynamic memory via the *malloc()* function, and reads the data into it.
3. Calculates the average of the points.
4. Returns the answer as a single floating-point value.

Since the amount of input and output for this program is explicitly known and because it reads all of its input at the beginning and writes all of its results at the end, a deadlock situation cannot occur.

The following IDL statements use *test_pipe* to determine the average of the values 0 to 9:

```
;Start test_pipe. The use of the NOSHELL keyword is not necessary,
;but speeds up the start-up process.
SPAWN, 'test_pipe', UNIT = UNIT, /NOSHELL

;Send the number of points followed by the actual data.
WRITEU, UNIT, 10L, FINDGEN(10)

;Read the answer.
READU, UNIT, ANSWER

;Announce the result.
PRINT, "Average = ", ANSWER

;Close the pipe, delete the child process, and deallocate the
;logical file unit.
FREE_LUN, UNIT
```

Executing these statements gives the result:

```
Average =       4.50000
```

This mechanism provides the UNIX IDL user a simple and efficient way to augment IDL with code written in other languages such as C or Fortran. It is, however, not as efficient as writing the required operation entirely in IDL. The actual cost depends primarily on the amount of data being transferred. For example, the above example can be performed entirely in IDL using a simple statement such as the following:

```
PRINT, 'Average = ', TOTAL(FINDGEN(10))/10.0
```

Chapter 3:

Overview: COM and ActiveX in IDL

This chapter discusses the following topics:

COM Objects and IDL

Microsoft's *Component Object Model*, or COM, is a specification for developing modular software components. COM is not a programming language or an API, but an implementation of a *component architecture*. A component architecture is a method of designing software components so that they can be easily connected together, reused, or replaced without re-compiling the application that uses them. Other examples of this methodology include the Object Management Group's *Common Object Request Broker Architecture* (CORBA) and Sun's *JavaBeans* technologies.

ActiveX controls are a special class of COM object that follow a set of Microsoft interface specifications; they are normally designed to present a user interface.

IDL for Windows supports three methods for using COM-based software components in your applications:

- Exposing a COM object as an IDL object,
- Including an ActiveX control in an IDL widget hierarchy,
- Including the IDLDrawWidget ActiveX control in an application written in a language other than IDL.

Note

While COM components can be developed for numerous platforms, most COM-based software is written for Microsoft Windows platforms. IDL for Windows supports the inclusion of COM technologies, but IDL for UNIX does not. The chapters in this section will discuss COM in the context of Microsoft Windows exclusively.

What are COM Objects?

A COM object, or *component,* is a piece of software that:

- is a library, rather than a standalone application (that is, it runs inside some sort of client application such as IDL, a Visual Basic application, or a web browser);
- is distributed in a compiled, executable form;
- exposes a group of methods and properties to its client application;

In addition to these criteria, a component may also supply a user interface that can be manipulated by the user. COM objects that supply a user interface and send *events* to the programs that use them are generally packaged as ActiveX controls, although it is not a requirement that an ActiveX control provide a user interface.

COM objects and ActiveX controls are nearly always packaged as Windows executable (`.exe`), dynamic link library(`.dll`), or object linking and embedding (`.ocx`) files.

Why Use COM Objects with IDL?

There are several reasons to use COM technologies alongside IDL:

- COM objects can be designed to use the facilities of the underlying Windows operating system. If you need access to Windows features not exposed within IDL, incorporating a COM object into your IDL program may provide the functionality you need.
- COM objects have been written to provide custom user interface elements or accomplish specific tasks. Many of these components are available to you free or at minimal cost. If you work exclusively in a Windows environment, incorporating a pre-written component in your IDL program may be faster than coding the same functionality in IDL.
- Using the IDLDrawWidget ActiveX control, you can rapidly incorporate IDL functionality into a Windows application created with any COM-aware environment. COM-aware environments include Visual Basic, Visual C++, and even VBScript.

Using COM Objects with IDL

The three methods for using COM objects with IDL are:

- Exposing a COM Object as an IDL Object,
- Including an ActiveX Control in an IDL Widget Hierarchy,
- Using the IDLDrawWidget ActiveX Control in an application written in a language other than IDL.

Exposing a COM Object as an IDL Object

IDL's IDLcomIDispatch object class creates an IDL object that communicates with an underlying COM object using the COM object's IDispatch interface. When you create an IDLcomIDispatch object, you provide the identifier for the COM object you wish to use, and IDL handles instantiation of and communication with the object. You can call the COM object's methods and get and set its properties using standard IDL object conventions and syntax.

Note

The IDLcomIDispatch object is useful when you want to incorporate a generic COM object into your IDL application. If the COM object you want to use is an ActiveX control, use the WIDGET_ACTIVEX routine, discussed below.

For details on using the IDLcomIDispatch object class to incorporate COM objects into your IDL applications, see Chapter 4, "Using COM Objects in IDL".

Including an ActiveX Control in an IDL Widget Hierarchy

IDL's WIDGET_ACTIVEX routine incorporates an ActiveX control directly into an IDL widget hierarchy. This allows you to place the ActiveX control in an IDL widget interface, and to receive widget events directly from the control for handling by a standard IDL widget event handler.

Internally, IDL uses the same mechanisms it uses when creating IDLcomIDispatch objects when it instantiates an ActiveX control as part of an IDL widget hierarchy. After the widget hierarchy has been realized, an object reference to the IDL object that encapsulates the ActiveX control can be retrieved and used as an interface with the ActiveX control. This allows you to call the ActiveX control's methods and get and set its properties using standard IDL object conventions and syntax.

For details on using the WIDGET_ACTIVEX routine to incorporate ActiveX controls into your IDL applications, see Chapter 5, "Using ActiveX Controls in IDL".

Using the IDLDrawWidget ActiveX Control

IDL for Windows distributions include an ActiveX control that makes IDL functionality available to other applications. Including the IDLDrawWidget control in your Windows application allows you to create your own user interface using the programming language of your choice, while using IDL's data analysis and display functionality.

Note

The IDLDrawWidget ActiveX control provides a COM interface to IDL, but requires an IDL installation to function. This means that in order for an application to use the IDLDrawWidget control, a licensed copy of IDL must be installed on the same computer.

For details on using the IDLDrawWidget ActiveX control in your own Windows applications, see Chapter 6, "The IDLDrawWidget ActiveX Control".

Skills Required to use COM Objects

Although IDL provides an abstracted interface to COM functionality, you must be familiar with some aspects of COM to successfully intertwine COM and IDL.

If You Are Using COM Objects

If you are using a COM object directly, via the IDLcomIDispatch object, you will need a thorough understanding of the COM object you are using, including its methods and properties. An understanding of the Windows tools used to discover information about COM objects is useful.

If You Are Using ActiveX Controls

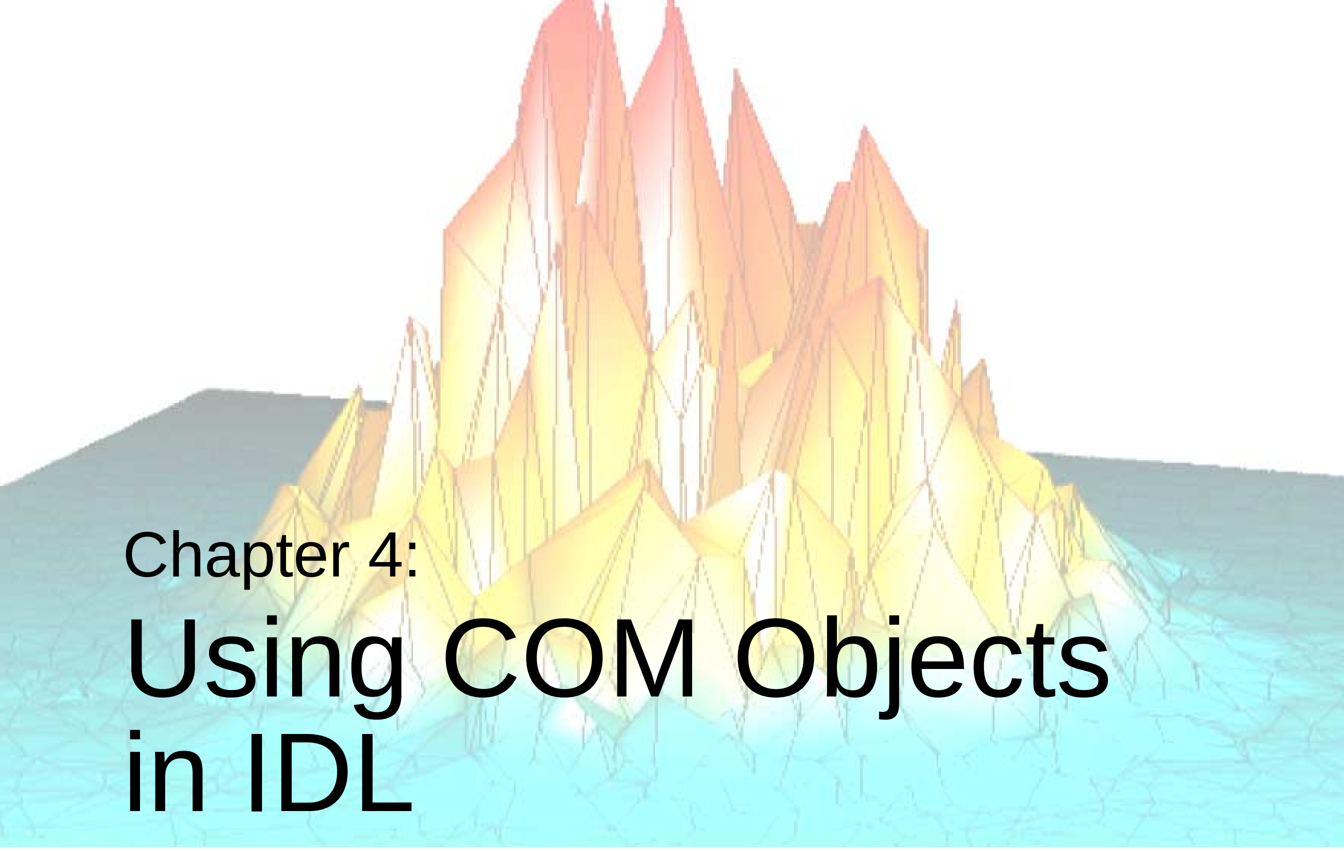
If you are incorporating an ActiveX control into an IDL widget hierarchy using WIDGET_ACTIVEX, you will need a thorough understanding of the ActiveX control you are using, including its methods, properties, and the information returned when an event is generated. An understanding of the Windows tools used to discover information about ActiveX controls is useful.

If You Are Using the IDLDrawWidget ActiveX Control

If you are incorporating the IDLDrawWidget ActiveX control in your own Windows application, you will need a thorough understanding of your own application development tools, including how they are used to interact with ActiveX controls. Details about the IDLDrawWidget control itself are provided in Chapter 6, “The IDLDrawWidget ActiveX Control” and Chapter 7, “IDLDrawWidget Control Reference”.

If You Are Creating Your Own COM Object

If you are creating your own COM object to be included in IDL, you will need a thorough understanding of both your development environment and of COM itself. It is beyond the scope of this manual to discuss creation of COM objects, but you should be able to incorporate any component created by following the COM specification into IDL by following the procedures outlined here.



Chapter 4: Using COM Objects in IDL

This chapter discusses the following topics:

Using COM Objects in IDL

If you want to incorporate a COM object that does not present its own user interface into your IDL application, use IDL's IDLcomIDispatch object class.

IDL's IDLcomIDispatch object class creates an IDL object that uses the COM IDispatch interface to communicate with an underlying COM object. When you create an IDLcomIDispatch object, you provide information about the COM object you wish to use, and IDL handles instantiation of and communication with the object. You can call the COM object's methods and get and set its properties using standard IDL object conventions and syntax.

Note

If the COM object you want to use in your IDL application is an ActiveX control, use the WIDGET_ACTIVEX routine, discussed in Chapter 5, "Using ActiveX Controls in IDL".

Object Creation

To create an IDL object that encapsulates a COM object, use the OBJ_NEW function as described in "Creating IDLcomIDispatch Objects" on page 54. IDL creates a dynamic subclass of the IDLcomIDispatch object class, based on information you specify for the COM object.

Method Calls and Property Management

Once you have created your IDLcomIDispatch object within IDL, use normal IDL object method calls to interact with the object. (See Chapter 22, "Object Basics" in the *Building IDL Applications* manual for a discussion of IDL objects.) COM object properties can be set and retrieved using the GetProperty and SetProperty methods implemented for the IDLcomIDispatch class. See "Method Calls on IDLcomIDispatch Objects" on page 55 and "Managing COM Object Properties" on page 63 for details.

Object Destruction

Destroy IDLcomIDispatch objects using the OBJ_DESTROY procedure. See "Destroying IDLcomIDispatch Objects" on page 66 for details.

Registering COM Components on a Windows Machine

Before a COM object or ActiveX control can be used by a client program, it must be *registered* on the Windows machine. In most cases, components are registered by the program that installs them on the machine. If you are using a component that is not installed by an installation program that handles the registration, you can register the component manually.

To register a component (`.dll` or `.exe`) or a control (`.ocx`), use the Windows command line program `regsvr32`, supplying it with name of the component or control to register. For example, the IDL distribution includes a COM component named RSIDemoComponent, contained in a file named `RSIDemoComponent.dll` located in the `examples\COMBridge` subdirectory of the IDL distribution. To register this component, do the following:

1. Open a Windows command prompt.
2. Change directories to the `examples\COMBridge` subdirectory of the IDL distribution.
3. Enter the following command:

```
regsvr32 RSIDemoComponent.dll
```

Windows will display a pop-up dialog informing you that the component has been registered. (You can specify the “ `/s` ” parameter to `regsvr32` to prevent the dialog from being displayed.)

Note

You only need to register a component once on a given machine. It is not necessary to register a component before each use.

IDLcomIDispatch Object Naming Scheme

When you create an IDLcomIDispatch object, IDL automatically creates a *dynamic subclass* of the IDLcomIDispatch class to contain the COM object. IDL determines which COM object to instantiate by parsing the class name you provide to the OBJ_NEW function. You specify the COM object to use by creating a class name that combines the name of the base class (IDLcomIDispatch) with either the COM class identifier or the COM program identifier for the object. The resulting class name looks like

```
IDLcomIDispatch$ID_type$ID
```

where *ID_type* is one of the following:

- `CLSID` if the object is identified by its COM class ID, or
- `PROGID` if the object is identified by its COM program ID,

and *ID* is the COM object's actual class or program identifier string.

Note

While COM objects incorporated into IDL are instances of the dynamic subclass created when the COM object is instantiated, they still expose the functionality of the class IDLcomIDispatch, which is the direct superclass of the dynamic subclass. All IDLcomIDispatch methods are available to the dynamic subclass.

Class Identifiers

A COM object's class identifier (generally referred to as the CLSID) is a 128-bit identifying string that is guaranteed to be unique for each object class. The strings used by COM as class IDs are also referred to as *Globally Unique Identifiers* (GUIDs) or *Universally Unique Identifiers* (UUIDs). It is beyond the scope of this chapter to discuss how class IDs are generated, but it is certain that every COM object has a unique CLSID.

COM class IDs are 32-character strings of alphanumeric characters and numerals that look like this:

```
{A77BC2B2-88EC-4D2A-B2B3-F556ACB52E52}
```

The above class identifier identifies the RSIDemoComponent class included with IDL.

When you create an IDLcomIDispatch object using a CLSID, you must modify the standard CLSID string in two ways:

1. You must omit the opening and closing braces ({ }).
2. You must replace the dash characters (-) in the CLSID string with underscores (_).

See "Creating IDLcomIDispatch Objects" on page 54 for example class names supplied to the OBJ_NEW function.

Note

If you do not know the class ID of the COM object you wish to expose as an IDL object, you may be able to determine it using an application provided by Microsoft; see "Finding COM Class and Program IDs" on page 52 for details.

Program Identifiers

A COM object's program identifier (generally referred to as the PROGID) is a mapping of the class identifier to a more human-friendly string. Unlike class IDs, program IDs are not guaranteed to be unique, so namespace conflicts are possible. Program IDs are, however, easier to work with; if you are not worried about name conflicts, use the identifier you are most comfortable with.

Program IDs are alphanumeric strings that can take virtually any form, although by convention they look like this:

```
PROGRAM.Component.version
```

For example, the RSIDemoComponent class included with IDL has the following program ID:

```
RSIDemoComponent.RSIDemoObj1.1
```

When you create an IDLcomIDispatch object using a PROGID, you must modify the standard PROGID string by replacing the dot characters (.) with underscores (_).

See "Creating IDLcomIDispatch Objects" on page 54 for example class names supplied to the OBJ_NEW function.

Note

If you do not know the program ID of the COM object you wish to expose as an IDL object, you may be able to determine it using an application provided by Microsoft; see "Finding COM Class and Program IDs" on page 52 for details.

Finding COM Class and Program IDs

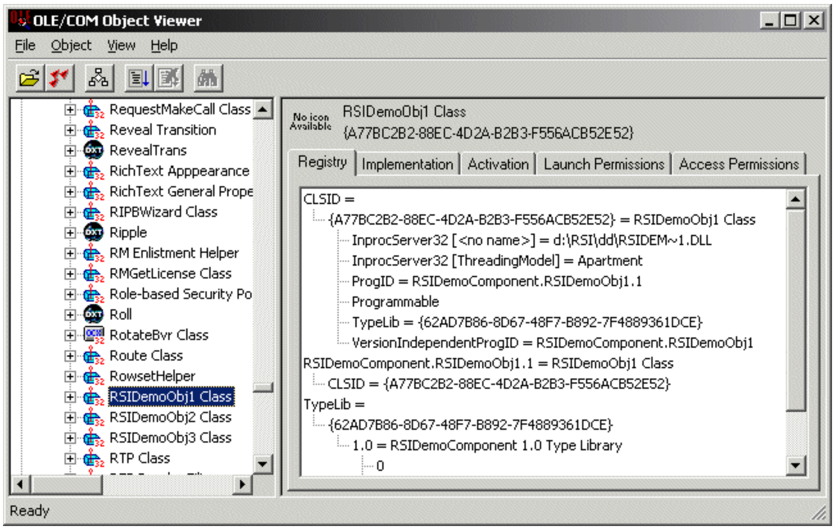
In general, if you wish to incorporate a COM object into an IDL program, you will know the COM class or program ID — either because you created the COM object yourself, or because the developer of the object provided you with the information.

If you do not know the class or program ID for the COM object you want to use, you may be able to determine them using the *OLE/COM Object Viewer* application provided by Microsoft. You can download the OLE/COM Object Viewer at no charge directly from Microsoft. As of this writing, you can locate the tool by pointing your Web browser to:

http://www.microsoft.com/com

and then selecting “Downloads” from the “Resources” menu.

The OLE/COM Object Viewer displays all of the COM objects installed on a computer, and allows you to view information about the objects and their interfaces.



RSIDemoObj1 Class
{A77BC2B2-88EC-4D2A-B2B3-F556ACB52E52}

```
CLSID =
  {A77BC2B2-88EC-4D2A-B2B3-F556ACB52E52} = RSIDemoObj1 Class
     InprocServer32 [<no name>] = d:\RSI\dd\RSIDEM~1.DLL
     InprocServer32 [ThreadingModel] = Apartment
     ProgID = RSIDemoComponent.RSIDemoObj1.1
     Programmable
     TypeLib = {62AD7B86-8D67-48F7-B892-7F4889361DCE}
     VersionIndependentProgID = RSIDemoComponent.RSIDemoObj1
RSIDemoComponent.RSIDemoObj1.1 = RSIDemoObj1 Class
  CLSID = {A77BC2B2-88EC-4D2A-B2B3-F556ACB52E52}
TypeLib =
  {62AD7B86-8D67-48F7-B892-7F4889361DCE}
     1.0 = RSIDemoComponent 1.0 Type Library
        0
```

Figure 4-1: Microsoft’s OLE/COM Object Viewer application.

Note

You can copy an object’s class ID to the clipboard by selecting the object in the leftmost panel of the object viewer, clicking the right mouse button, and selecting “Copy CLSID to Clipboard” from the context menu.

If you have an IDL program that instantiates a COM object running on your computer, you can determine either the class ID or the program ID by using the HELP command with the OBJECTS keyword. IDL displays the full dynamic subclass name, including the class ID or program ID that was used when the object was created.

Creating IDLcomIDispatch Objects

To expose a COM object as an IDL object, use the OBJ_NEW function to create a dynamic subclass of the IDLcomIDispatch object class. The name of the subclass must be constructed as described in "IDLcomIDispatch Object Naming Scheme" on page 50, and identifies the COM object to be instantiated.

Note

If the COM object you want to use within IDL is an ActiveX control, use the WIDGET_ACTIVEX routine as described in Chapter 5, "Using ActiveX Controls in IDL". Instantiating the ActiveX control as part of an IDL widget hierarchy allows you to respond to events generated by the control, whereas COM objects that are instantiated using the OBJ_NEW do not generate events that IDL is aware of.

For example, suppose you wish to include a COM component with the class ID

```
{A77BC2B2-88EC-4D2A-B2B3-F556ACB52E52}
```

and the program ID

```
RSIDemoComponent.RSIDemoObj1.1
```

in an IDL program. Use either of the following calls to the OBJ_NEW function:

```
ObjRef = OBJ_NEW( $
   'IDLcomIDispatch$CLSID$A77BC2B2_88EC_4D2A_B2B3_F556ACB52E52')
```

or

```
ObjRef = OBJ_NEW( $
   'IDLcomIDispatch$PROGID$RSIDemoComponent_RSIDemoObj1_1')
```

IDL's internal COM subsystem instantiates the COM object within an IDLcomIDispatch object with one of the following the dynamic class names

```
IDLcomIDispatch$CLSID$A77BC2B2_88EC_4D2A_B2B3_F556ACB52E52
```

or

```
IDLcomIDispatch$PROGID$RSIDemoComponent_RSIDemoObj1_1
```

and sets up communication between the object and IDL. You can work with the IDLcomIDispatch object just as you would with any other IDL object; calling the object's methods, and getting and setting its properties.

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

Method Calls on IDLcomIDispatch Objects

IDL allows you to call the underlying COM object's methods by calling methods on the IDLcomIDispatch object. IDL handles conversion between IDL data types and the data types used by the component, and any results are returned in IDL variables of the appropriate type.

As with all IDL objects, the general syntax is:

```
result = ObjRef -> Method([Arguments])
```

or

```
ObjRef -> Method[, Arguments]
```

where `ObjRef` is an object reference to an instance of a dynamic subclass of the IDLcomIDispatch class.

Function vs. Procedure Methods

In COM, all object methods are *functions*. IDL's implementation of the IDLcomIDispatch object maps COM methods that supply a return value using the `retval` attribute as IDL functions, and COM methods that do not supply a return value via the `retval` attribute as procedures. See "Displaying Interface Information using the Object Viewer" on page 59 for more information on determining which methods use the `retval` attribute.

The IDLcomIDispatch::GetProperty and IDLcomIDispatch::SetProperty methods are special cases. These methods are IDL object methods — not methods of the underlying COM object — and they use *procedure* syntax. The process of getting and setting properties on COM objects encapsulated in IDLcomIDispatch objects is discussed in "Managing COM Object Properties" on page 63.

Note

The IDL object system uses method names to identify and call object lifecycle methods (Init and Cleanup). If the COM object underlying an IDLcomIDispatch object implements Init or Cleanup methods, they will be overridden by IDL's lifecycle methods — the COM object's methods will be inaccessible from IDL. Similarly, IDL implements the GetProperty and SetProperty methods for the IDLcomIDispatch object, so any methods of the underlying COM object that use these names will be inaccessible from IDL.

What Happens When a Method Call is Made?

When a method is called on an IDLcomIDispatch object, the method name and arguments are passed to the internal IDL COM subsystem, where they are used to construct the appropriate IDispatch method calls for the underlying COM object.

From the point of view of an IDL user issuing method calls on the IDLcomIDispatch object, this process is completely transparent. The IDL user simply calls the COM object's method using IDL syntax, and IDL handles the translation.

Data Type Conversions

IDL and COM use different data types internally. While you should be aware of the types of data expected by the COM object's methods and the types it returns, you do not need to worry about converting between IDL data types and COM data types manually. IDL's dynamic type conversion facilities handle all conversion of data types between IDL and the COM system. The data type mappings are described in "COM-IDL Data Type Mapping" on page 67.

For example, if the COM object that underlies an IDLcomIDispatch object has a method that requires a value of type INT as an input argument, you would supply the value as an IDL Long. If you supplied the value as any other IDL data type, IDL would first convert the value to an IDL Long using its normal data type conversion mechanism before passing the value to the COM object as an INT.

Similarly, if a COM object returns a BOOL value, IDL will place the value in a variable of Byte type, with a value of 1 (one) signifying True or a value of 0 (zero) signifying False.

Optional Arguments

Like IDL routines, COM object methods can have *optional arguments*. Optional arguments eliminate the need for the calling program to provide input data for all possible arguments to the method for each call. The COM optional argument functionality is passed along to COM object methods called on IDLcomIDispatch objects, and to the IDLcomIDispatch::GetProperty method. This means that if an argument is not required by the underlying COM object method, it can be omitted from the method call used on the IDLcomIDispatch object.

Note

Only method arguments defined with the `optional` token in the object's interface definition are optional. See "Displaying Interface Information using the Object

Viewer" on page 59 for more information regarding the object's interface definition file.

Warning

If an argument that is *not* optional is omitted from the method call used on the IDLcomIDispatch object, IDL will generate an error.

Argument Order

Like IDL, COM treats arguments as *positional parameters*. This means that it makes a difference where in the argument list an argument occurs. (Contrast this with IDL's handling of keywords, which can occur anywhere in the argument list after the routine name.) COM enforces the following ordering for arguments to object methods:

1. Required arguments
2. Optional arguments for which default values are defined
3. Optional arguments for which no default values are defined

The same order applies when the method is called on an IDLcomIDispatch object.

Default Argument Values

COM allows objects to specify a default value for any method arguments that are optional. If a call to a method that has an optional argument with a default value omits the optional argument, the default value is used. IDL behaves in the same way as COM when calling COM object methods on IDLcomIDispatch objects, and when calling the IDLcomIDispatch::GetProperty method.

Method arguments defined with the `defaultvalue()` token in the object's interface definition are optional, and will use the specified default value if omitted from the method call. See "Displaying Interface Information using the Object Viewer" on page 59 for more information regarding the object's interface definition file.

Argument Skipping

COM allows methods with optional arguments to accept a subset of the full argument list by specifying which arguments are not present. This allows the calling routine to supply, for example, the first and third arguments to a method, but not the second. IDL provides the same functionality for COM object methods called on IDLcomIDispatch objects, but not for the IDLcomIDispatch::GetProperty or SetProperty methods.

To skip one or more arguments from a list of optional arguments, include the SKIP keyword in the method call. The SKIP keyword accepts either a scalar or a vector of numbers specifying which arguments are not provided.

Note

The indices for the list of method arguments are zero-based — that is, the first method argument (either optional or required) is argument 0 (zero), the next is argument 1 (one), *etc*.

For example, suppose a COM object method accepts four arguments, of which the second, third, and fourth are optional:

```
ObjMethod, arg1, arg2-optional, arg3-optional, arg4-optional
```

To call this method on the IDLcomIDispatch object that encapsulates the underlying COM object, skipping `arg2`, use the following command:

```
objRef -> ObjMethod, arg1, arg3, arg4, SKIP=1
```

Note that the SKIP keyword uses the index value 1 to indicate the second argument in the argument list. Similarly, to skip `arg2` and `arg3`, use the following command:

```
objRef -> ObjMethod, arg1, arg4, SKIP=[1,2]
```

Finally, note that you do not need to supply the SKIP keyword if the arguments are supplied in order. For example, to skip `arg3` and `arg4`, use the following command:

```
objRef -> ObjMethod, arg1, arg2
```

Finding Object Methods

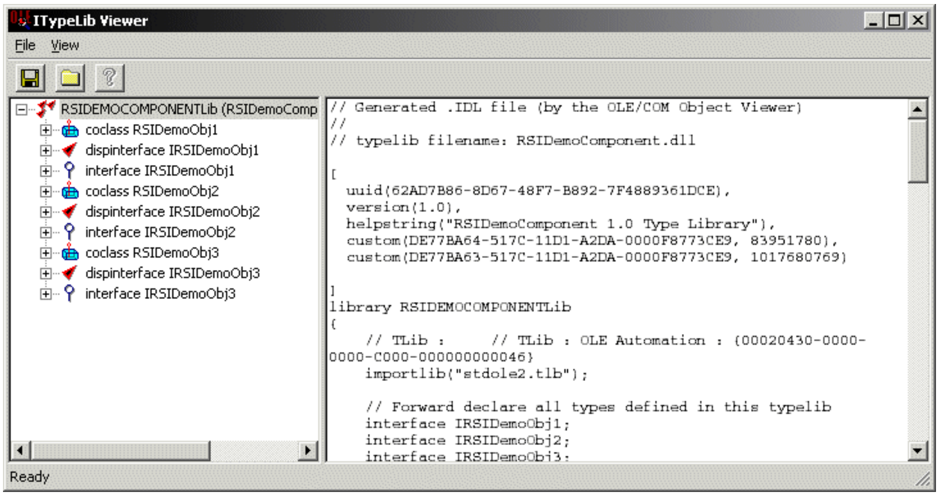
In most cases, when you incorporate a COM object into an IDL program, you will know what the COM object's methods are and what arguments and data types those methods take — either because you created the COM object yourself, or because the developer of the object provided you with the information. If for some reason you do not know what methods the COM object supports, you may be able to determine which methods are available and what parameters they accept using the *OLE/COM Object Viewer* application provided by Microsoft. (See "Finding COM Class and Program IDs" on page 52 for information on acquiring the OLE/COM Object Viewer.)

Warning

Finding information about a COM object's methods using the OLE/COM Object Viewer requires a moderately sophisticated understanding of COM programming, or at least COM interface definitions. While we provide some hints in this section on how to interpret the interface definition, if you are not already familiar with the structure of COM objects you may find this material inadequate. If possible, consult the developer of the COM object you wish to use rather than attempting to determine its structure using the object viewer.

Displaying Interface Information using the Object Viewer

You can use the OLE/COM Object Viewer to view the interface definitions for any COM object on your Windows machine. Select a COM object in the leftmost panel of the object viewer, click the right mouse button, and select “View Type Information...” A new window titled “ITypeLib Viewer” will be displayed, showing all of the component’s interfaces (Figure 4-2).



```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: RSIDemoComponent.dll

[
  uuid(62AD7B86-8D67-48F7-B892-7F4889361DCE),
  version(1.0),
  helpstring("RSIDemoComponent 1.0 Type Library"),
  custom(DE77BA64-517C-11D1-A2DA-0000F8773CE9, 83951780),
  custom(DE77BA63-517C-11D1-A2DA-0000F8773CE9, 1017680769)

]
library RSIDEMOCOMPONENTLib
{
    // TLib :     // TLib : OLE Automation : {00020430-0000-
0000-C000-000000000046}
    importlib("stdole2.tlb");

    // Forward declare all types defined in this typelib
    interface IRSIDemoObj1;
    interface IRSIDemoObj2;
    interface IRSIDemoObj3;
```

Figure 4-2: Viewing a COM object’s interface definition.

Note

The top lines in the right-hand panel will say something like:

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: RSIDemoComponent.dll
```

The “.IDL file” in this case has nothing to do with IDL, the Interactive Data Language. Here “IDL” stands for *Interface Description Language* — a language used to define component interfaces. If you are familiar with the Interface

Description Language, you can often determine what a component is designed to do.

With the top-level object selected in the left-hand pane of the ITypelib Viewer, scroll down in the right-hand pane until you find the section that defines the IDispatch interface for the object in question. The definition will look something like this:

```
interface IRSIDemoObj1 : IDispatch {
   [id(0x00000001)]
   HRESULT GetCLSID([out, retval] BSTR* pBstr);
   [id(0x00000002), propput]
   HRESULT MessageStr([in] BSTR pstr);
   [id(0x00000002), propget]
   HRESULT MessageStr([out, retval] BSTR* pstr);
   [id(0x00000003)]
   HRESULT DisplayMessageStr();
   [id(0x00000004)]
   HRESULT Msg2InParams(
      [in] BSTR str,
      [in] long val,
      [out, retval] BSTR* pVal);
   [id(0x00000005)]
   HRESULT GetIndexObject(
      [in] long ndxObj,
      [out, retval] IDispatch** ppDisp);
   [id(0x00000006)]
   HRESULT GetArrayOfObjects(
      [out] long* pObjCount,
      [out, retval] VARIANT* psaObjs);
};
```

Method definitions look like this:

```
[id(0x00000001)]
HRESULT GetCLSID([out, retval] BSTR* pBstr);
```

where the line including the `id` string is an identifier used by the object to refer to its methods and the following line or lines (usually beginning with `HRESULT`) define the method’s interface.

Again, while it is beyond the scope of this manual to discuss COM object methods in detail, the following points may assist you in determining how to use a COM object:

- Methods whose definitions include the `retval` attribute will appear in IDL as functions.

  ```
  [id(0x00000001)]
  HRESULT GetCLSID([out, retval] BSTR* pBstr);
  ```

- Methods that do not include the `retval` attribute will appear in IDL as procedures.

```
[id(0x00000003)]
HRESULT DisplayMessageStr();
```

- Methods whose definitions include the `propget` attribute allow you to retrieve an object property using the IDLcomIDispatch::GetProperty method. You cannot call these methods directly in IDL; see "Managing COM Object Properties" on page 63 for additional details.

```
[id(0x00000002), propget]
HRESULT MessageStr([out, retval] BSTR* pstr);
```

- Methods whose definitions include the `propput` attribute allow you to set an object property using the IDLcomIDispatch::SetProperty method. You cannot call these methods directly in IDL; see "Managing COM Object Properties" on page 63 for additional details.

```
[id(0x00000002), propput]
HRESULT MessageStr([in] BSTR pstr);
```

- Methods that accept optional input values will include the `optional` token in the argument's definition. For example, the following definition indicates that the second input argument is optional:

```
[id(0x00000004)]
HRESULT Msg1or2InParams(
   [in] BSTR str,
   [in, optional] int val,
   [out, retval] BSTR* pVal);
```

- Methods that provide default values for optional arguments replace the `optional` token with the `defaultvalue()` token, where the default value of the argument is supplied between the parentheses. For example, the following definition indicates that the second input argument is optional, and has a default value of 15:

```
HRESULT Msg1or2InParams(
   [in] BSTR str,
   [in, defaultvalue(15)] int val,
   [out, retval] BSTR* pVal);
```

- While methods generally return an `HRESULT` value, this is not a requirement.

Displaying Interface Information using the IDL HELP Procedure

If you have an IDL program that instantiates a COM object running on your computer, you can determine either the class ID or the program ID by using the

HELP command with the OBJECTS keyword. IDL displays a list of objects, along with their methods, with function and procedure methods in separate groups for each object class.

Managing COM Object Properties

As a convenience to the IDL programmer, COM object methods that have been defined using the `propget` and `propput` attributes are accessible via the IDLcomIDispatch object's GetProperty and SetProperty methods. This means that rather than calling the COM object's methods directly to get and set property values, you use the standard IDL syntax.

Note

If a COM object method's interface definition includes either the `propget` or the `propput` attribute, you *must* use the IDL GetProperty and SetProperty methods to get and set values. IDL does not allow you to call these methods directly.

As with all IDL objects, the IDLcomIDispatch object's GetProperty and SetProperty methods use procedure syntax. Keywords to the methods represent the names of the properties being retrieved or set, and the keyword values represent either an IDL variable into which the property value is placed or an IDL expression that is the value to which the property is set. You must use the procedure syntax when calling either method, even if the underlying COM object methods being used are functions rather than procedures.

Setting Properties

To set a property value on a COM object, use the following syntax:

```
ObjRef -> SetProperty, KEYWORD=Expression
```

where `ObjRef` is the IDLcomIDispatch object that encapsulates the COM object, *KEYWORD* is the COM object property name, and *Expression* is an IDL expression representing the property value to be set.

You can set multiple property values in a single statement by supplying multiple *KEYWORD=Expression* pairs.

Note

KEYWORD must map exactly to the full name of the underlying COM object's property setting method. The partial keyword name functionality provided by IDL is not valid with IDLcomIDispatch objects.

Getting Properties

To retrieve a property value from a COM object, use the following syntax:

```
ObjRef -> GetProperty, KEYWORD=Variable
```

where `ObjRef` is the IDLcomIDispatch object that encapsulates the COM object, *KEYWORD* is the COM object property name, and *Variable* is the name of an IDL variable that will contain the retrieved property value.

You can get multiple property values in a single statement by supplying multiple *KEYWORD=Variable* pairs.

Note

KEYWORD must map exactly to the full name of the underlying COM object's property getting method. The partial keyword name functionality provided by IDL is not valid with IDLcomIDispatch objects.

Because some of the underlying COM object's `propget` methods may require arguments, the IDLcomIDispatch object's GetProperty method will accept optional arguments. To retrieve a property using a method that takes arguments, use the following syntax:

```
ObjRef -> GetProperty, KEYWORD=Variable [, arg0, arg1, ... argn]
```

Note, however, that if arguments are required, you can only specify one property to retrieve.

References to Other COM Objects

It is not uncommon for COM objects to return references to other COM objects, either as a property value or via an object method. If an IDLcomIDispatch object returns a reference to another COM object's IDispatch interface, IDL automatically creates an IDLcomIDispatch object to contain the object reference.

For example, suppose the `GetOtherObject` method to the COM object encapsulated by the IDLcomIDispatch object `Obj1` returns a reference to another COM object.

```
Obj2 = Obj1 -> GetOtherObject()
```

Here, `Obj2` is an IDLcomIDispatch object that encapsulates some other COM object. `Obj2` behaves in the same manner as any IDLcomIDispatch object.

Note that IDLcomIDispatch objects created in this manner are not linked in any way to the object whose method created them. In the above example, this means that destroying `Obj1` does not destroy `Obj2`. If the COM object you are using creates new IDLcomIDispatch objects in this manner, you must be sure to explicitly destroy the automatically-created objects along with those you explicitly create, using the OBJ_DESTROY procedure.

Destroying IDLcomIDispatch Objects

Use the OBJ_DESTROY procedure to destroy and IDLcomIDispatch object.

When OBJ_DESTROY is called with an IDLcomIDispatch object as an argument, the underlying reference to the COM object is released and IDL resources relating to that object are freed.

Note

Destroying an IDLcomIDispatch object does not automatically cause the destruction of the underlying COM object. COM employs a reference-counting methodology and expects the COM object to destroy itself when there are no remaining references. When an IDLcomIDispatch object is destroyed, IDL simply decrements the reference count on the underlying COM object.

COM-IDL Data Type Mapping

When data moves from IDL to a COM object and back, IDL handles conversion of variable data types automatically. The data type mappings are shown in Table 4-1.

COM Type	IDL Type
BOOL (VT_BOOL)	Byte (true =1, false=0)
ERROR (VT_ERROR)	Long
CY (VT_CY)	Double (see note below)
DATE (VT_DATE)	Double
I1 (VT_I1)	Byte
INT (VT_INT)	Long
UINT (VT_UINT)	Unsigned Long
VT_USERDEFINED	The IDL type is passed through.
VT_UI1	Byte
VT_I2	Integer
VT_UI2	Unsigned integer
VT_ERROR	Long
VT_I4	Long
VT_UI4	Unsigned Long
VT_I8	Long64
VT_UI8	Unsigned Long 64
VT_R4	Float
VT_BSTR	String
VT_R8	Double
VT_DISPATCH	IDLcomIDispatch

Table 4-1: IDL-COM Data Type Mapping.

COM Type	IDL Type
VT_UNKNOWN	IDLcomIDispatch

Table 4-1: (Continued) IDL-COM Data Type Mapping.

Note on the COM CY Data Type

The COM CY data type is a scaled 64-bit integer, supporting exactly four digits to the right of the decimal point. To provide an easy-to-use interface, IDL automatically scales the integer as part of the data conversion that takes place between COM and IDL, allowing the IDL user to treat the number as a double-precision floating-point value. When the value is passed back to the COM object, it will be truncated if there are more than four significant digits to the right of the decimal point.

For example, the IDL double-precision value `234.56789` would be passed to the COM object as `234.5678`.

Example: RSIDemoComponent

This example uses a COM component included in the IDL distribution. The RSIDemoComponent is included purely for demonstration purposes, and does not perform any useful work beyond illustrating how IDLcomIDispatch objects are created and used.

The RSIDemoComponent is contained in a file named `RSIDemoComponent.dll` located in the `examples\COMBridge` subdirectory of the IDL distribution. Before attempting to execute this example, make sure the component is registered on your system as described in "Registering COM Components on a Windows Machine" on page 49.

There are three objects defined by the RSIDemoComponent. The example begins by using RSIDemoObj1, which has the program ID:

```
RSIDemoComponent.RSIDemoObj1
```

and the class ID:

```
{A77BC2B2-88EC-4D2A-B2B3-F556ACB52E52}
```

Note

The following section develops an IDL procedure called IDispatchDemo that illustrates use of the RSIDemoComponent. The complete `.pro` file is included in the `examples\COMBridge` subdirectory of the IDL distribution as `IDispatchDemo.pro`.

1. Begin by creating an IDLcomIDispatch object from the COM object. You can use either the class ID or the program ID. Remember that if you use the class ID, you must remove the braces ({ }) and replace the hyphens with underscores.

   ```
   obj1 = OBJ_NEW( $
      'IDLCOMIDispatch$PROGID$RSIDemoComponent_RSIDemoObj1')
   ```

   or (with Class ID):

   ```
   obj1 = OBJ_NEW( $
      'IDLCOMIDispatch$CLSID$A77BC2B2_88EC_4D2A_B2B3_F556ACB52E52')
   ```

2. The COM object implements the `GetCLSID` method, which returns the class ID for the component. You can retrieve this value in and IDL variable and print it. The string should be `'{A77BC2B2-88EC-4D2A-B2B3-F556ACB52E52}'`.

   ```
   strCLSID = obj1->GetCLSID()
   PRINT, strCLSID
   ```

Note

The GetCLSID method returns the class identifier of the object using the standard COM separators (`-`).

3. The COM object has a property named `MessageStr`. To retrieve the value of the `MessageStr` property, enter:

   ```
   obj1 -> GetProperty, MessageStr = outStr
   PRINT, outStr
   ```

   IDL should print `'RSIDemoObj1'`.

4. You can also set the `MessageStr` property of the object and display it using the object's `DisplayMessageStr` method, which displays the value of the `MessageStr` property in a Windows dialog:

   ```
   obj1 -> SetProperty, MessageStr = 'Hello, world'
   obj1 -> DisplayMessageStr
   ```

5. The `Msg2InParams` method takes two input parameters and concatenates them into a single string. Executing the following commands should cause IDL to print `'The value is: 25'`.

   ```
   instr = 'The value is: '
   val = 25L
   outStr = obj1->Msg2InParams(instr, val)
   PRINT, outStr
   ```

6. To view all known information about the IDLcomIDispatch object, including its dynamic subclass name and the names of its methods, use the IDL HELP command with the OBJECTS keyword:

   ```
   HELP, obj1, /OBJECTS
   ```

7. The `GetIndexObject()` method returns an object reference to one of the following three possible objects:

   - `RSIDemoObj1` (index = 1)
   - `RSIDemoObj2` (index = 2)
   - `RSIDemoObj3` (index = 3)

Note

If the index is not 1, 2, or 3, the `GetIndexObject` method will return an error.

To get a reference to `RSIDemoObj3`, use the following command:

```
obj3 = obj1->GetIndexObject(3)
```

8. All three objects have the `GetCLSID` method. You can use this method to verify that the desired object was returned. The output of the following commands should be `'{13AB135D-A361-4A14-B165-785B03AB5023}'`.

   ```
   obj3CLSID = obj3->GetCLSID()
   PRINT, obj3CLSID
   ```

9. Remember to destroy a retrieved object when you are finished with it:

   ```
   OBJ_DESTROY, obj3
   ```

10. Next, use the COM object's `GetArrayOfObjects()` method to return a vector of object references to `RSIDemoObj1`, `RSIDemoObj2`, and `RSIDemoObj3`, respectively. The number of elements in the vector is returned in the first parameter; the result should `3`.

    ```
    objs = obj1->GetArrayOfObjects(cItems)
    PRINT, cItems
    ```

11. Since each object implements the `GetCLSID` method, you could loop through all the object references and get its class ID:

    ```
    FOR i = 0, cItems-1 do begin
       objCLSID = objs[i] -> GetCLSID()
       PRINT, 'Object[',i,'] CLSID: ', objCLSID
    ENDFOR
    ```

12. Remember to destroy object references when you are finished with them:

    ```
    OBJ_DESTROY, objs
    OBJ_DESTROY, obj1
    ```

Chapter 5:

Using ActiveX Controls in IDL

This chapter discusses the following topics:

Using ActiveX Controls in IDL

If you want to incorporate a COM object that presents a user interface (that is, an ActiveX control) into your IDL application, use IDL's WIDGET_ACTIVEX routine to place the control in an IDL widget hierarchy. IDL provides the same object method and property manipulation facilities for ActiveX controls as it does for COM objects incorporated using the IDLcomIDispatch object interface, but adds the ability to process events generated by the ActiveX control using IDL's widget event handling mechanisms.

Note

IDL can only incorporate ActiveX controls on Windows NT/2000/XP (and later) platforms.

When you use the WIDGET_ACTIVEX routine, IDL automatically creates an IDLcomActiveX object that encapsulates the ActiveX control. IDLcomActiveX objects are a subclass of the IDLcomIDispatch object class, and share all of the IDLcomIDispatch methods and mechanisms discussed in Chapter 4, "Using COM Objects in IDL". You should be familiar with the material in that chapter before attempting to incorporate ActiveX controls in your IDL programs.

Note

If the COM object you want to use in your IDL application is *not* an ActiveX control, use the IDLcomIDispatch object class.

Registering COM Components on a Windows Machine

Before a COM object or ActiveX control can be used by a client program, it must be *registered* on the Windows machine. In most cases, components are registered by the program that installs them on the machine. If you are using a component that is not installed by an installation program that handles the registration, you can register the component manually. For a description of the registration process, see "Registering COM Components on a Windows Machine" on page 49.

ActiveX Control Naming Scheme

When you incorporate an ActiveX control into an IDL widget hierarchy using the WIDGET_ACTIVEX routine, IDL automatically creates an IDLcomActiveX object that instantiates the control and handles all communication between it and IDL. You tell IDL which ActiveX control to instantiate by passing the COM class or program ID for the ActiveX control to the WIDGET_ACTIVEX routine as a parameter.

IDL automatically creates a *dynamic subclass* of the IDLcomActiveX class (which is itself a subclass of the IDLcomIDispatch class) to contain the ActiveX control. The resulting class name looks like

```
IDLcomActiveX$ID_type$ID
```

where *ID_type* is one of the following:

- `CLSID` if the object is identified by its COM class ID, or
- `PROGID` if the object is identified by its COM program ID,

and *ID* is the COM object's actual class or program identifier string.

For more on COM class and program IDs see "Class Identifiers" on page 50 and "Program Identifiers" on page 51.

While you will never need to use this dynamic class name directly, you may see it reported by IDL via the HELP routine or in error messages. Note that when IDL reports the name of the dynamic subclass, it will replace the hyphen characters in a class ID and the dot characters in a program ID with underscore characters. This is because neither the hyphen nor the dot character are valid in IDL object names.

Finding COM Class and Program IDs

In general, if you wish to incorporate an ActiveX object into an IDL widget hierarchy, you will know the COM class or program ID — either because you created the control yourself, or because the developer of the control provided you with the information.

If you do now know the class or program ID for the COM object you want to use, you may be able to determine them using the *OLE/COM Object Viewer* application provided by Microsoft. For more information, see "Finding COM Class and Program IDs" on page 52.

Creating ActiveX Controls

To include an ActiveX control in an IDL application, use the WIDGET_ACTIVEX function, supplying the COM class or program ID of the ActiveX control as the *COM_ID* argument.

Note

If the object you want to use in your IDL application is *not* an ActiveX control, use the IDLcomIDispatch object class as described in Chapter 4, “Using COM Objects in IDL”. Instantiating a non-ActiveX component using the WIDGET_ACTIVEX function is not supported, and may lead to unpredictable results.

Once the ActiveX object has been instantiated within an IDL widget hierarchy, you can call the control’s native methods as described in “Method Calls on ActiveX Controls” on page 78, and access or modify its properties as described in “Managing ActiveX Control Properties” on page 80. IDL widget events generated by the control are discussed in “ActiveX Widget Events” on page 81.

For example, suppose you wished to include an ActiveX control with the class ID:

```
{0002E510-0000-0000-C000-000000000046}
```

and the program ID:

```
OWC.Spreadsheet.9
```

in an IDL widget hierarchy. Use either of the following calls the WIDGET_ACTIVEX function:

```
wAx = WIDGET_ACTIVEX(wBase, $
   '0002E510-0000-0000-C000-000000000046')
```

or

```
wAx = WIDGET_ACTIVEX(wBase, 'OWC.Spreadsheet.9', ID_TYPE=1)
```

where `wBase` is the widget ID of the base widget that will contain the ActiveX control.

Note

When instantiating an ActiveX control using the WIDGET_ACTIVEX function, you do not need to modify the class or program ID as you do when creating an IDLcomIDispatch object using the OBJ_NEW function. Be aware, however, that when IDL creates the underlying IDLcomActiveX object, the dynamic class name will replace the hyphens from a class ID or the dots from a program ID with underscore characters.

IDL’s internal COM subsystem instantiates the ActiveX control within an IDLcomActiveX object with one of the following dynamic class names

```
IDLcomActiveX$CLSID$0002E510_0000_0000_C000_000000000046
```

or

```
IDLcomActiveX$PROGID$OWC_Spreadsheet_9
```

and sets up communication between the object and IDL. IDL also places the control into the specified widget hierarchy and prepares to accept widget events generated by the control.

See “WIDGET_ACTIVEX” in the *IDL Reference Guide* manual for additional details.

Method Calls on ActiveX Controls

IDL allows you to call the underlying ActiveX control's methods by calling methods on the IDLcomActiveX object that is automatically created when you call the WIDGET_ACTIVEX function. IDL handles conversion between IDL data types and the data types used by the component, and any results are returned in IDL variables of the appropriate type.

As with all IDL objects, the general syntax is:

```
result = ObjRef -> Method([Arguments])
```

or

```
ObjRef -> Method[, Arguments]
```

where `ObjRef` is an object reference to an instance of a dynamic subclass of the IDLcomActiveX class.

The IDLcomActiveX object class is a direct subclass of the IDLcomIDispatch object class and provides none of its own methods. As a result, method calls on IDLcomActiveX objects follow the same rules as calls on IDLcomIDispatch objects. You should read and understand "Method Calls on IDLcomIDispatch Objects" on page 55 before calling an ActiveX control's methods.

Retrieving the Object Reference

Unlike IDLcomIDispatch objects, which you create explicitly with a call to the OBJ_NEW function, IDLcomActiveX objects are created automatically by IDL. To obtain an object reference to the automatically created IDLcomActiveX object, use the GET_VALUE keyword to the WIDGET_CONTROL procedure.

For example, consider the following lines of IDL code:

```
wBase = WIDGET_BASE()
wAx = WIDGET_ACTIVEX(wBase, 'myProgram.myComponent.1', ID_TYPE=1)
WIDGET_CONTROL, wBase, /REALIZE
WIDGET_CONTROL, wAx, GET_VALUE=oAx
```

The first line creates a base widget that will hold the ActiveX control. The second line instantiates the ActiveX control using its program ID and creates an IDLcomActiveX object. The third line realizes the base widget and the ActiveX control it contains — note that the ActiveX widget must be realized before you can retrieve a reference to the IDLcomActiveX object. The fourth line uses the WIDGET_CONTROL procedure to retrieve an object reference to the

IDLcomActiveX object in the variable `oAx`. You can use this object reference to call the ActiveX control's methods and set its properties.

Managing ActiveX Control Properties

As a convenience to the IDL programmer, ActiveX control methods that have been defined using the `propget` and `propput` attributes are accessible via the IDLcomActiveX object's GetProperty and SetProperty methods, which are inherited directly from the IDLcomIDispatch object class. This means that rather than calling the ActiveX control's methods directly to get and set property values, you use the standard IDL syntax.

The IDLcomActiveX object class is a direct subclass of the IDLcomIDispatch object class and provides none of its own methods. As a result, IDL's facilities for managing the properties of ActiveX controls follow the same rules as for IDLcomIDispatch objects. You should read and understand "Managing COM Object Properties" on page 63 before working with an ActiveX control's properties.

ActiveX Widget Events

Events generated by an ActiveX control are dispatched using the standard IDL widget methodology. When an ActiveX event is passed into IDL, it is packaged in an anonymous IDL structure that contains the ActiveX event parameters.

While the actual structure of an event generated by an ActiveX control will depend on the control itself, the following gives an idea of the structure's format:

```
{ID             : 0L,
 TOP            : 0L,
 HANDLER        : 0L,
 DISPID         : 0L, ; The DISPID of the callback method
 EVENT_NAME     : "", ; The name of the callback method
<Param1 name>  : <Param1 value>,
<Param2 name>  : <Param2 value>,

<ParamN name>  : <ParamN value>
}
```

As with other IDL Widget event structures, the first three fields are standard. ID is the widget id of the widget generating the event, TOP is the widget ID of the top level widget containing ID, and HANDLER contains the widget ID of the widget associated with the handler routine.

The DISPID field contains the decimal representation of the *dispatch ID* (or DISPID) of the method that was called. Note that in the OLE/COM Object Viewer, this ID number is presented as a *hexadecimal* number. Other applications (Microsoft Visual Studio among them) may display the decimal representation.

The EVENT_NAME field contains the name of the method that was called.

The *Param name* fields contain the values of parameters returned by the called method. The actual parameter name or names displayed, if any, depend on the method being called by the ActiveX control.

Using the ActiveX Widget Event Structure

Since the widget event structure generated by an ActiveX control depends on the method that generated the event, it is important to check the type of event before processing values in IDL. Successfully parsing the event structure requires a detailed understanding of the dispatch interface of the ActiveX control; you must know either the DISPID or the method name of the method, and you must know the names and data types of the values returned.

For example, suppose the ActiveX control you are incorporating into your IDL application includes two methods named `Method1` and `Method2` in a dispatch interface that looks like this:

```
dispinterface MyDispInterface {
    properties:
    methods:
      [id(0x00000270)]
      void Method1([in] EventInfo* EventInfo);
      [id(0x00000272)]
      HRESULT Method2([out, retval] BSTR* EditData);
};
```

A widget event generated by a call to `Method1`, which has no return values, would look something like:

```
** Structure <3fb7288>, 5 tags, length=32, data length=32:
   ID               LONG           13
   TOP              LONG           12
   HANDLER          LONG           12
   DISPID           LONG          624
   EVENT_NAME       STRING     'Method1'
```

Note that the DISPID is 624, the decimal equivalent of 270 hexadecimal.

A widget event generated by a call to `Method2`, which has one return value, would look something like:

```
** Structure <3fb7288>, 6 tags, length=32, data length=32:
   ID               LONG           13
   TOP              LONG           12
   HANDLER          LONG           12
   DISPID           LONG          626
   EVENT_NAME       STRING     'Method2'
   EDITDATA         STRING     'some text value'
```

An IDL event-handler routine could use the value of the DISPID field to check which of these two ActiveX control methods generated the event before attempting to use the value of the EDITDATA field:

```
PRO myRoutine_event, event
   IF(event.DISPID eq 626) THEN BEGIN
      PRINT, event.EDITDATA
   ENDIF ELSE BEGIN
      <do something else>
   ENDELSE
END
```

Dynamic Elements in the ActiveX Event Structure

Parameter data included in an event structure generated by an ActiveX control can take the form of an array. If this happens, the array is placed in an IDL pointer, and the pointer, rather than the array itself, is included in the IDL event structure. Similarly, an ActiveX control may return a reference to another COM object, as described in “References to Other COM Objects” on page 65, in its event structure.

IDL pointers and objects created in this way are not automatically removed; it is the IDL programmer’s responsibility free them using a routine such as PTR_FREE, HEAP_FREE, or OBJ_DESTROY.

If it is unclear whether the event structure will contain dynamic elements (objects or pointers) it is best to pass the ActiveX event structure to the HEAP_FREE routine when your event-handler routine has finished with the event. This will ensure that all dynamic portions of the structure are released.

Destroying ActiveX Controls

An ActiveX control incorporated in an IDL widget hierarchy is destroyed when any of the following occurs:

- When the widget hierarchy to which the ActiveX widget belongs is destroyed.
- When a call to WIDGET_CONTROL, *wAx*, /DESTROY is made, where *wAx* is the widget ID of the ActiveX widget.
- When the underlying IDLcomActiveX object is destroyed by a call to OBJ_DESTROY.

In most cases, cleanup of an application that includes an ActiveX control is not different from an application using only IDL native widgets. However, because it is possible for an ActiveX control to return references to other COM objects to IDL, you must be sure to keep track of all objects created by your application and destroy them as necessary. See "References to Other COM Objects" on page 65 for details.

In addition, it is possible for the widget event structure generated by an ActiveX control to include IDL pointers or object references. Pointers and object references included in the event structure are not automatically destroyed. See "Dynamic Elements in the ActiveX Event Structure" on page 83 for more information.

Example: Calendar Control

This example uses an ActiveX control that displays a calendar interface. The control, contained in the file `mscal.ocx`, is installed along with a typical installation of Microsoft Office 97, and may also be present on your system if you have upgraded to a more recent version of Microsoft Office. If the control is *not* present on your system (you'll know the control is not present if the example code does not display a calendar similar to the one shown in Figure 5-1 on page 87), you can download a the control as part of a package of sample ActiveX controls included in the file `actxsamp.exe`, discussed in Microsoft Knowledge Base Article 165437.

If you download the control, place the file `mscal.exe` in a known location and execute the file; you will be prompted for a directory in which to place `mscal.ocx`. Open a command prompt window in the directory you chose and register the control as described in "Registering COM Components on a Windows Machine" on page 49.

The calendar control has the program ID:

```
MSCAL.Calendar.7
```

and the class ID:

```
{8E27C92B-1264-101C-8A2F-040224009C02}
```

Note

The following section develops an IDL routine called ActiveXCal that illustrates use of the calendar ActiveX control within an IDL widget hierarchy. The complete `.pro` file is included in the `examples\COMBridge` subdirectory of the IDL distribution as `ActiveXCal.pro`.

1. Create the ActiveXCal procedure. (Remember that in the `ActiveXCal.pro` file, this procedure occurs last.)

   ```
   PRO ActiveXCal
   ```

2. Create a top-level base widget to hold the ActiveX control.

   ```
   wBase = WIDGET_BASE(COLUMN = 1, SCR_XSIZE = 400, $
      TITLE='IDL ActiveX Widget Calendar Control')
   ```

3. Create base widgets to hold labels for the selected month, day, and year. Set the initial values of the labels.

   ```
   wSubBase = WIDGET_BASE(wBase, /ROW)
   wVoid = WIDGET_LABEL(wSubBase, value = 'Month: ')
   wMonth = WIDGET_LABEL(wSubBase, value = 'October')
   wSubBase = WIDGET_BASE(wBase, /ROW)
   wVoid = WIDGET_LABEL(wSubBase, VALUE = 'Day: ')
   ```

```
      wDay = WIDGET_LABEL(wSubBase, VALUE = '22')
      wSubBase = WIDGET_BASE(wBase, /ROW)
      wVoid = WIDGET_LABEL(wSubBase, VALUE = 'Year: ')
      wYear = WIDGET_LABEL(wSubBase, VALUE = '1999')
```

4. Instantiate the ActiveX Control, using the control's class ID.

```
      wAx=WIDGET_ACTIVEX(wBase, $
         '{8E27C92B-1264-101C-8A2F-040224009C02}')
```

5. Realize the top-level base widget.

```
      WIDGET_CONTROL, wBase, /REALIZE
```

6. Set the top-level base's user value to an anonymous structure containing widget IDs of the month, day, and year label widgets.

```
      WIDGET_CONTROL, wBase, $
         SET_UVALUE = {month:wMonth, day:wDay, year:wYear}
```

7. Retrieve the object ID of the IDLcomActiveX object that encapsulates the ActiveX control. Use the GetProperty method to retrieve the current values of the month, day, and year from the control.

```
      WIDGET_CONTROL, wAx, GET_VALUE = oAx
      oAx->GetProperty, month=month, day=day, year=year
```

8. Set the values of the label widgets to reflect the current date, as reported by the ActiveX control.

```
      WIDGET_CONTROL, wMonth, SET_VALUE=STRTRIM(month, 2)
      WIDGET_CONTROL, wDay, SET_VALUE=STRTRIM(day, 2)
      WIDGET_CONTROL, wYear, SET_VALUE=STRTRIM(year, 2)
```

9. Call XMANAGER to manage the widget events, and end the procedure.

```
      XMANAGER, 'ActiveXCal', wBase

   END
```

10. Now create an event-handling routine for the calendar control. (Remember that in the `ActiveXCal.pro` file, this procedure occurs before the ActiveXCal procedure.)

```
   PRO ActiveXCal_event, ev
```

11. The ActiveX widget is the only widget in this application that generates widget events, so the ID field of the event structure is guaranteed to contain the widget ID of that widget. Use the GET_VALUE keyword to retrieve an object reference to the IDLcomActiveX object that encapsulates the control.

```
      WIDGET_CONTROL, ev.ID, GET_VALUE = oCal
```

12. The user value of the top-level base widget is an anonymous structure that holds the widget IDs of the month, day, and year label widgets (see step 6 above). Retrieve the structure into a variable named `state`.

```
WIDGET_CONTROL, ev.TOP, GET_UVALUE = state
```

13. Use the GetProperty method on the IDLcomActiveX object to retrieve the current values of the month, day, and year from the calendar control.

```
ocal->GetProperty, month=month, day=day, year=year
```

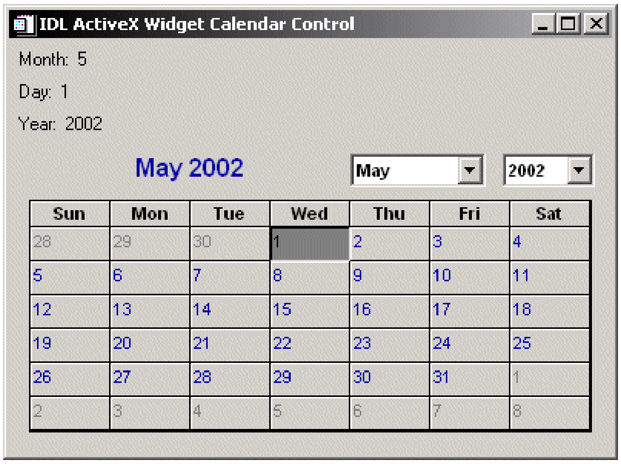
14. Use WIDGET_CONTROL to set the values of the month, day, and year label widgets.

```
WIDGET_CONTROL, state.month, SET_VALUE = STRTRIM(month,2)
WIDGET_CONTROL, state.day, SET_VALUE = STRTRIM(day,2)
WIDGET_CONTROL, state.year, SET_VALUE = STRTRIM(year,2)
```

15. Call HEAP_FREE to ensure that dynamic portions of the event structure are released, and end the procedure.

```
    HEAP_FREE, ev

END
```

Running the ActiveXCal procedure displays a widget that looks like the following:



Month: 5
Day: 1
Year: 2002

Sun	Mon	Tue	Wed	Thu	Fri	Sat
28	29	30	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1
2	3	4	5	6	7	8

Figure 5-1: An IDL widget program using an ActiveX calendar control.

Example: Spreadsheet Control

This example uses an ActiveX control that displays a spreadsheet interface. The control, contained in the file `msowc.dll`, is installed along with a typical installation of Microsoft Office. If the control is *not* present on your system (you'll know the control is not present if the example code fails when trying to realize the widget hierarchy), the example will not run.

The spreadsheet control has the program ID:

```
OWC.Spreadsheet.9
```

and the class ID:

```
{0002E510-0000-0000-C000-000000000046}
```

Information about the spreadsheet control's properties and methods was gleaned from *Microsoft Excel 97 Visual Basic Step by Step* by Reed Jacobson (Microsoft Press, 1997) and by inspection of the control's interface using the *OLE/COM Object Viewer* application provided by Microsoft. It is beyond the scope of this manual to describe the spreadsheet control's interface in detail.

Note

The following section develops an IDL routine called ActiveXSS that illustrates use of the spreadsheet ActiveX control within an IDL widget hierarchy. The complete `.pro` file is included in the `examples\COMBridge` subdirectory of the IDL distribution as `ActiveXSS.pro`.

1. Create a function that will retrieve data from cells selected in the spreadsheet control. The function takes two arguments: an object reference to the IDLcomActiveX object that instantiates the spreadsheet control, and a variable to contain the data from the selected cells.

   ```
   FUNCTION ss_getSelection, oSS, aData
   ```

2. Retrieve an object that represents the selected cells. Note that when the ActiveX control returns this object, IDL automatically creates an IDLcomActiveX object that makes it accessible within IDL.

   ```
      oSS->GetProperty, SELECTION=oSel
   ```

3. Retrieve the total number of cells selected.

   ```
      oSel->GetProperty, COUNT=nCells
   ```

4. If no cells are selected, destroy the selection object and return zero (the failure code).

```
IF (nCells LT 1) THEN BEGIN
   OBJ_DESTROY, oSel
   RETURN, 0
ENDIF
```

5. Retrieve objects that represent the dimensions of the selection.

```
oSel->GetProperty, COLUMNS=oCols, ROWS=oRows
```

6. Get the dimensions of the selection, then destroy the column and row objects.

```
oCols->GetProperty, COUNT=nCols
OBJ_DESTROY, oCols
oRows->GetProperty, COUNT=nRows
OBJ_DESTROY, oRows
```

7. Create a floating point array with the same dimensions as the selection.

```
aData = FLTARR(nCols, nRows, /NOZERO);
```

8. Iterate through the cells, doing the following:

   - Retrieve an object that represents the cell. Note that the numeric index of the FOR loop is passed to the GetProperty method as an argument.
   - Get the value contained in the cell.
   - Set the appropriate element of the aData array to the cell's value.
   - Destroy the object.

```
FOR i=1, nCells DO BEGIN
   oSel->GetProperty, ITEM=oItem, i
   oItem->GetProperty, VALUE=vValue
   aData[i-1] = vValue
   OBJ_DESTROY, oItem
ENDFOR
```

9. Destroy the selection object.

```
OBJ_DESTROY, oSel
```

10. Return one (the success code) and end the function definition.

```
   RETURN, 1

END
```

11. Next, create a procedure that sets the values of the cells in the spreadsheet. This procedure takes one argument: an object reference to the IDLcomActiveX object that instantiates the spreadsheet control.

```
PRO ss_setData, oSS
```

12. Define the size of the data array.

```
   nX = 20
```

13. Get an object representing the active spreadsheet.

```
   oSS->GetProperty, ActiveSheet=oSheet
```

14. Get an object representing the cells in the spreadsheet.

```
   oSheet->GetProperty, CELLS=oCells
```

15. Generate some data.

```
   im = BESELJ(DIST(nX))
```

16. Iterate through the elements of the data array, doing the following:

- Retrieve an object that represents the cell that corresponds to the data element. Note that the numeric indices of the FOR loops are passed to the GetProperty method as arguments.
- Set the value of the cell.
- Destroy the object.

```
   FOR i=0, nX-1 DO BEGIN
      FOR j=0, nX-1 DO BEGIN
         oCells->GetProperty, ITEM=oItem, i+1, j+1
         oItem->SetProperty, VALUE=im(i,j)
         OBJ_DESTROY, oItem
      ENDFOR
   ENDFOR
```

17. Destroy the spreadsheet and cell objects, and end the procedure.

```
   OBJ_DESTROY, oSheet
   OBJ_DESTROY, oCells

END
```

18. Next, create a procedure to handle events generated by the widget application.

```
PRO ActiveXSS_event, ev
```

19. The user value of the top-level base widget is set equal to a structure that contains the widget ID of the ActiveX widget. Retrieve the structure into the variable sState.

```
WIDGET_CONTROL, ev.TOP, GET_UVALUE=sState, /NO_COPY
```

20. Use the value of the DISPID field of the event structure to sort out “selection changing” events.

```
IF (ev.DISPID EQ 1513) THEN BEGIN
```

21. Place data from selected cells in variable aData, using the `ss_getSelection` function defined above. Check to make sure that the function returns a success value (one) before proceeding.

```
  IF (ss_getSelection(sState.oSS, aData) NE 0) THEN BEGIN
```

22. Get the dimensions of the aData variable.

```
     szData = SIZE(aData)
```

23. If aData is two-dimensional, display a surface, otherwise, plot the data.

```
     IF (szData[0] GT 1 AND szData[1] GT 1 AND szData[2] GT 1) $
       THEN SURFACE, aData $
     ELSE $
       PLOT, aData
  ENDIF

ENDIF
```

24. Reset the state variable sState and end the procedure.

```
WIDGET_CONTROL, ev.TOP, SET_UVALUE=sState, /NO_COPY

END
```

25. Create the main widget creation routine.

```
PRO ActiveXSS

   !EXCEPT=0  ; Ignore floating-point underflow errors.
```

26. Create a top-level base widget.

```
wBase = WIDGET_BASE(COLUMN=1, $
   TITLE="IDL ActiveX Spreadsheet Example")
```

27. Instantiate the ActiveX spreadsheet control in a widget.

```
wAx=WIDGET_ACTIVEX(wBase, $
   '{0002E510-0000-0000-C000-000000000046}', $
   SCR_XSIZE=600, SCR_YSIZE=400)
```

28. Realize the widget hierarchy.

```
WIDGET_CONTROL, wBase, /REALIZE
```

29. The value of an ActiveX widget is an object reference to the IDLcomActiveX object that encapsulates the ActiveX control. Retrieve the object reference in the variable oSS.

```
WIDGET_CONTROL, wAx, GET_VALUE=oSS
```

30. Turn off the TitleBar property on the spreadsheet control.

```
oSS->SetProperty, DisplayTitleBar=0
```

31. Populate the spreadsheet control with data, using the `ss_setData` function defined above.

```
ss_setData, oSS
```

32. Set the user value of the top-level base widget to an anonymous structure that contains the widget ID of the spreadsheet ActiveX widget.

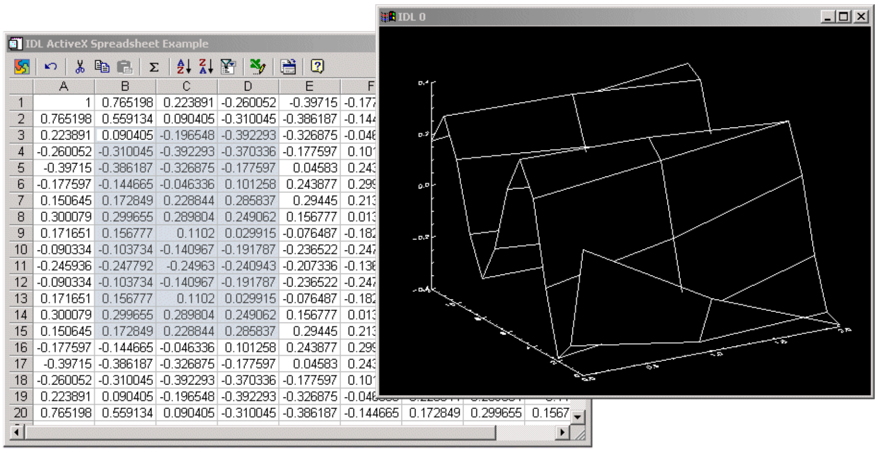
```
WIDGET_CONTROL, wBase, SET_UVALUE={oSS:oSS}
```

33. Call XMANAGER to manage the widgets, and end the procedure.

```
XMANAGER,'ActiveXSS', wBase, /NO_BLOCK
```

```
END
```

Running the ActiveXSS procedure display widgets that look like the following:



	A	B	C	D	E	F			
1	1	0.765198	0.223891	-0.260052	-0.39715	-0.177			
2	0.765198	0.559134	0.090405	-0.310045	-0.386187	-0.144			
3	0.223891	0.090405	-0.196548	-0.392293	-0.326875	-0.046			
4	-0.260052	-0.310045	-0.392293	-0.370336	-0.177597	0.101			
5	-0.39715	-0.386187	-0.326875	-0.177597	0.04583	0.243			
6	-0.177597	-0.144665	-0.046336	0.101258	0.243877	0.299			
7	0.150645	0.172849	0.228844	0.285837	0.29445	0.213			
8	0.300079	0.299655	0.289804	0.249062	0.156777	0.013			
9	0.171651	0.156777	0.1102	0.029915	-0.076487	-0.182			
10	-0.090334	-0.103734	-0.140967	-0.191787	-0.236522	-0.247			
11	-0.245936	-0.247792	-0.24963	-0.240943	-0.207336	-0.136			
12	-0.090334	-0.103734	-0.140967	-0.191787	-0.236522	-0.247			
13	0.171651	0.156777	0.1102	0.029915	-0.076487	-0.182			
14	0.300079	0.299655	0.289804	0.249062	0.156777	0.013			
15	0.150645	0.172849	0.228844	0.285837	0.29445	0.213			
16	-0.177597	-0.144665	-0.046336	0.101258	0.243877	0.299			
17	-0.39715	-0.386187	-0.326875	-0.177597	0.04583	0.243			
18	-0.260052	-0.310045	-0.392293	-0.370336	-0.177597	0.101			
19	0.223891	0.090405	-0.196548	-0.392293	-0.326875	-0.046			
20	0.765198	0.559134	0.090405	-0.310045	-0.386187	-0.144665	0.172849	0.299655	0.1567

Figure 5-2: An IDL widget program using an ActiveX spreadsheet control.

Chapter 6: The IDLDrawWidget ActiveX Control

This chapter discusses the following topics:

Overview

The Microsoft Windows version of IDL includes an ActiveX control that provides a powerful way to integrate all the data analysis and visualization features of IDL with other programming languages that support ActiveX controls. ActiveX is a set of technologies that enables software components to interact, regardless of the language in which they were written. This makes it possible, for example, to design a software interface with Microsoft Visual Basic and have IDL respond to the events it generates. The major features of the IDL ActiveX control include the following:

- The IDL ActiveX control makes it possible to display IDL direct and object graphics within an OLE container that supports ActiveX controls;
- The IDL ActiveX control can respond to events, regardless of whether they are generated by an external program or IDL itself;
- The IDL ActiveX control greatly simplifies the process of moving data to and from IDL and an external program;
- And finally, the interface to the IDL ActiveX control appears native to the external application.

Other issues to note regarding the ActiveX control are:

- The IDL ActiveX control is intended primarily for use in applications developed with Visual Basic 5.0 or greater. The control can be included in any programming language designed to use ActiveX controls (e.g. Visual C++ or Delphi). Users who intend to utilize the IDL ActiveX control in Visual C++ applications should be thoroughly familiar with Microsoft Foundation Classes and ActiveX programming. The IDL ActiveX control uses Visual Basic-style data types to exchange data between a Visual Basic application and IDL. A Visual C++ programmer will need to use OLE's `VARIANT` and `SAFEARRAY` types. A discussion of how to use the IDL ActiveX control with these languages is beyond the scope of this manual.
- The IDL ActiveX control does not support any non-blocking IDL widgets. When you call a widget from an ActiveX Control, you will not have access to the active command line and control will not pass back to the calling program until the blocking has been removed (the widget has been dismissed). You can, however, recreate the functionality of a widget using the given functionality. For an example, see "XLoadCT Functionality Using Visual Basic" on page 110.

The ActiveX interface to IDL consists of a single control called **IDLDrawWidget**. When this control is included in a project, it exposes the features of IDL through its properties and methods. The **IDLDrawWidget** can also trigger events. The properties and methods of the **IDLDrawWidget** are listed in Chapter 7, "IDLDrawWidget Control Reference".

In this chapter, you will be guided through a series of examples designed to demonstrate techniques for integrating IDL with programs written in Microsoft Visual Basic. These techniques begin with writing a simple application that shows how IDL can respond to Visual Basic events and draw graphics in a Visual Basic window.

A Note about Versions of the IDL ActiveX Control

Periodically, RSI releases a new version of the IDLDrawX ActiveX control. Older versions of the control will continue to work as they always have, but the new versions may include new features or other enhancements.

Why Are New Versions of the Control Created?

One of the features of COM is that interfaces are immutable. That is to say that when you create an interface, you "contractually" agree that the interface won't change. Changes to the way the control interacts with other components require that a new interface — and thus a new version of the control — must be created. Since the IDL ActiveX control is a COM object it is bound by this agreement. When RSI makes improvements to the ActiveX control interface by adding new methods and properties, we release a new ActiveX control with the new interface.

What Must You Change to Take Advantage of a New Control?

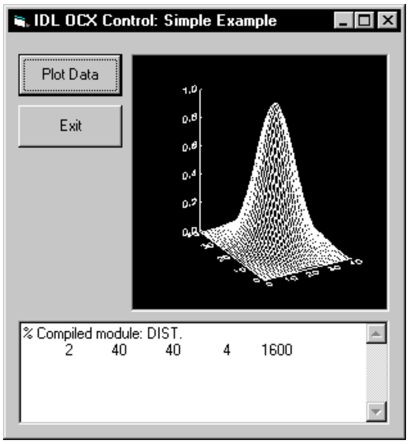
If you are a Visual Basic user, you need to add the new version of the control to your project and remove the old versions. For example, if you are upgrading to the "IDLDrawX3 ActiveX Control Module" included with IDL version 5.6, you would add this control to your project and remove the "IDLDrawX ActiveX Control Module" or "IDLDrawX2 ActiveX Control Module" from your project. The source code need not change.

What About Previous ActiveX Controls?

While previous versions of the IDLDrawX control will continue to work with new versions of IDL, they are no longer supported and will not be shipped with IDL. It is recommended that you upgrade to the new version to take advantage of new features and bug fixes.

Creating an Interface and Handling Events

The goal of this first example is very simple: to create a user interface in Microsoft Visual Basic and have IDL respond to events and display an image. The following figure shows what the finished project looks like when it runs. The Visual Basic source code used to create the example is shown in the following figure:



```
% Compiled module: DIST.
       2      40      40       4    1600
```

Figure 6-1: A simple example showing the IDLDrawWidget and text returned by IDL

As the figure shows, our first example program consists of two buttons ("Plot Data" and "Exit"), a graphics area, and a text box. All of these elements reside on top of what is called a form in Visual Basic parlance. (A form in Visual Basic is similar to a top level base in IDL.) Clicking the "Plot Data" button causes IDL to produce the surface plot shown. Clicking “Exit” causes IDL and the Visual Basic program to free memory and exit.

Visual Basic

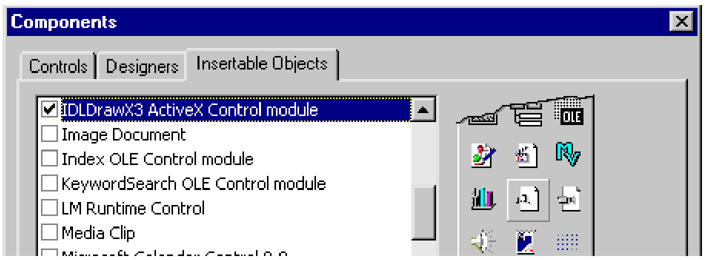
```
Private Sub Form_Load()
    n = IDLDrawWidget1.InitIDL(Form1.hWnd)
    If n <= 0 Then
        MsgBox ("IDL failed to initialize")
        End
    End If
    IDLDrawWidget1.CreateDrawWidget
    IDLDrawWidget1.SetOutputWnd (IDL_Output_Box.hWnd)
End Sub

Private Sub Plot_Button_Click()
    IDLDrawWidget1.ExecuteStr ("Z = SHIFT(DIST(40), 20, 20)")
    IDLDrawWidget1.ExecuteStr ("Z = EXP(-(Z/10)^2)")
    IDLDrawWidget1.ExecuteStr ("SURFACE, Z")
    IDLDrawWidget1.ExecuteStr ("PRINT, SIZE(Z)")
End Sub

Private Sub Exit_Button_Click()
    IDLDrawWidget1.DoExit
    End
End Sub
```

Table 6-1: Source code for a simple example

Drawing the Interface

Begin building the first example by creating a new Visual Basic project, adding the IDL ActiveX control, and drawing the interface components. Launch Microsoft Visual Basic and create a new project.

1. Add the IDL ActiveX component to the project. Visual Basic displays a list of all available components when you select the Components from the Project menu.



☑ IDLDrawX3 ActiveX Control module
☐ Image Document
☐ Index OLE Control module
☐ KeywordSearch OLE Control module
☐ LM Runtime Control
☐ Media Clip

Figure 6-2: List of Available Components

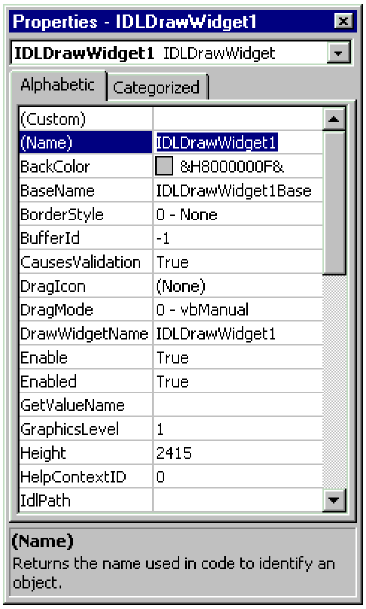
Select the “IDLDrawX3 ActiveX Control module” check box and close the Components window. Visual Basic will display the IDLDrawWidget’s icon in the toolbar, as shown to the left.

2. Begin drawing the interface. The “Plot” and “Exit” buttons were created with the **CommandButton** widget, the text box was created with the **TextBox** widget, and the graphics display area was created with **IDLDrawWidget**.

Specifying the IDL Path and Graphics Level

Having added **IDLDrawWidget** to the Visual Basic project, we now have access to **IDLDrawWidget**’s properties and methods. Use the **IdlPath** and **GraphicsLevel** properties to specify the directory path of the IDL ActiveX control and to choose between IDL’s direct and object graphics capabilities. Refer to Chapter 7, “IDLDrawWidget Control Reference” for a complete list of the properties and methods to **IDLDrawWidget**.

1. Use Visual Basic’s Properties window to select the **IDLDrawWidget**. All of the **IDLDrawWidget**’s properties can be set using the Properties window. Many properties can also be set within the source code. These distinctions are noted in Chapter 7, “IDLDrawWidget Control Reference”.



(Custom)	
(Name)	IDLDrawWidget1
BackColor	&H8000000F&
BaseName	IDLDrawWidget1Base
BorderStyle	0 - None
BufferId	-1
CausesValidation	True
DragIcon	(None)
DragMode	0 - vbManual
DrawWidgetName	IDLDrawWidget1
Enable	True
Enabled	True
GetValueName	
GraphicsLevel	1
Height	2415
HelpContextID	0
IdlPath	

(Name)
Returns the name used in code to identify an object.

Figure 6-3: Visual Basic Properties window

2. Locate the **IdlPath** property and enter the directory path to your IDL installation. If you installed IDL in its default location, this path will be:

   ```
   c:\rsi\idl54
   ```

3. Locate the **GraphicsLevel** property and set it equal to **1**. This selects IDL's direct graphics. A setting of 2 selects IDL's object graphics.

Initializing IDL

With the interface drawn and the properties of the **IDLDrawWidget** set, now write some Visual Basic code to give the application behavior. By double-clicking on the form which contains all of the interface components, Visual Basic will automatically generate the following subroutine.

```
Private Sub Form_Load()
End Sub
```

Visual Basic's **Form_Load** routine executes automatically when a program starts running. This procedure can be used to initialize IDL, create the **IDLDrawWidget**, and direct output from IDL to a text box. The code to accomplish these tasks will be placed between the two statements listed above.

IDL needs to be initialized before Visual Basic can interact with the **IDLDrawWidget**. This is done with the **InitIDL** method. **InitIDL** takes the **hWnd** of the form containing the **IDLDrawWidget** as an argument and returns 1 or less than 1, depending on whether or not IDL initialized successfully. Assuming that the default names given to the form and the **IDLDrawWidget** were not changed, IDL can be initialized with the following statement.

```
n = IDLDrawWidget1.InitIDL(Form1.hWnd)
```

A conditional statement is included to display an error message and exit the program if IDL failed to initialize.

```
If n <= 0 Then
    MsgBox ("IDL failed to initialize")
    End
End If
```

Creating the Draw Widget

When a box is drawn with the “IDLDrawWidget” icon in the toolbar, an OCX frame is created. This is a container for the **IDLDrawWidget**. This container is analogous to an IDL widget base. The graphics window that will be used by IDL still must be created. This is accomplished with the **CreateDrawWidget** method, as shown in the following statement:

```
IDLDrawWidget1.CreateDrawWidget
```

Directing IDL Output to a Text Box

The example program displays any output returned by IDL in a text box created in Visual Basic. This is accomplished with the **SetOutputWnd** method of the **IDLDrawWidget**. The **SetOutputWnd** method takes the **hWnd** of the text box that will contain the IDL output as an argument. The text box in the example program is named **IDL_Output_Box**, hence the following statement.

```
IDLDrawWidget1.SetOutputWnd (IDL_Output_Box.hWnd)
```

Note

Although this is the last statement within the **Form_Load()** subroutine, it could be placed before the call to **InitIDL** to get standard IDL version information printed.

Responding to Events and Issuing IDL Commands

The easiest way to integrate IDL with Visual Basic is to let Visual Basic manage the events and pass instructions to IDL. Recall that our example program contains two buttons: “Plot Data” and “Exit”. When you double-click on “Plot Data”, Visual Basic automatically creates the following subroutine:

```
Private Sub Plot_Button_Click()
End Sub
```

Visual Basic will execute any statements within this subroutine when the user clicks “Plot Data”. Instructions are passed to IDL using the **ExecuteStr** method to the **IDLDrawWidget**. The **ExecuteStr** method takes a string as an argument. This string is passed to IDL for execution as if it were entered at the IDL command line. The five statements which follow instruct IDL to produce the surface plot shown in the figure above.

```
IDLDrawWidget1.ExecuteStr ("Z = SHIFT(DIST(40), 20, 20)")
IDLDrawWidget1.ExecuteStr ("Z = EXP(-(Z/10)^2)")
IDLDrawWidget1.ExecuteStr ("SURFACE, Z")
IDLDrawWidget1.ExecuteStr ("PRINT, SIZE(Z)")
```

Cleaning Up and Exiting

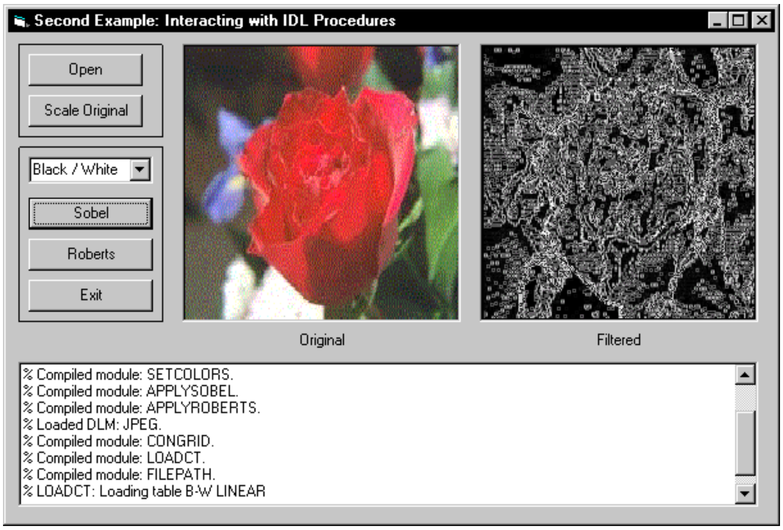
This project exits when the user clicks “Exit”. Exiting is a two step process. IDL is given a chance to clean up and exit by issuing the **DoExit** method. The Visual Basic program then exits with an **End** statement.

```
Private Sub Exit_Button_Click()
    IDLDrawWidget1.DoExit
    End
End Sub
```

Working with IDL Procedures

In this next example a project is created that uses multiple IDL procedures. Here the same issues apply as when developing a standard IDL program with a graphical user interface. In addition, managing memory when moving from one procedure to another should be considered. It is important to realize that the ActiveX control interacts with IDL at the main level. Thus, a Visual Basic program passing instructions to IDL is identical to entering the same instructions at the IDL command line. In this example Visual Basic is only used to create the user interface and dispatch events. The data resides in memory controlled by IDL. IDL is used for all data processing and display functions.

The following figure shows the user interface of the example project. The project is part of the IDL distribution and resides in the `examples\doc\ActiveX\SecondExample` directory.



```
% Compiled module: SETCOLORS.
% Compiled module: APPLYSOBEL.
% Compiled module: APPLYROBERTS.
% Loaded DLM: JPEG.
% Compiled module: CONGRID.
% Compiled module: LOADCT.
% Compiled module: FILEPATH.
% LOADCT: Loading table B-W LINEAR
```

Figure 6-4: The User Interface with Two Draw Widgets

The user interface consists of two **IDLDrawWidget** objects. The one on the left will display an image read from a *JPEG* file. The window on the right displays what the image looks like after processing. Buttons allow the user to scale the image and perform Roberts and Sobel filtering operations on the data.

Creating the Interface

The interface is created as it was in the first example, by drawing the interface components in Visual Basic. Two **IDLDrawWidget**s are created. Set the path `(c:\rsi\idl54)` and graphics level properties (type 1) of both.

Initializing IDL

Although there are two **IDLDrawWidget** objects, only one instance of the ActiveX control needs to be initialized. Both of the **IDLDrawWidget** objects do need to be created, however.

This is done with the two statements below:

```
IDLDrawWidget1.CreateDrawWidget
IDLDrawWidget2.CreateDrawWidget
```

Compiling the IDL Code

This example uses IDL procedures contained in a `.pro` file named `SecondExample.pro`. This file contains IDL procedures. Before these procedures can be called from Visual Basic, `SecondExample.pro` needs to be compiled. This assumes that the `.pro` file resides in the same directory as the Visual Basic project. The path method of the *App* object returns the directory from which the Visual Basic application was launched. Pass this directory to IDL with the statements

```
WorkingDirectory = "CD, ’" + App.Path + "’"
IDLDrawWidget1.ExecuteStr (WorkingDirectory)
```

The `.pro` can then be compiled. A conditional statement is used to exit the program if IDL was unable to locate the `.pro` file.

Dispatching Button Events to IDL

Because Visual Basic is used primarily for the user interface components of the application, IDL’s procedures have been created for processing the button events in the application. This is accomplished through the **ExecuteStr** method of the

IDLDrawWidget, as called in the following figure; when you click “Open”, the **OpenFile** procedure is defined as below.

Visual Basic

```
Private Sub Open_Button_Click(Index As Integer)
    IDLCommand = "OpenFile, " + Str(BaseID)
    IDLDrawWidget1.ExecuteStr (IDLCommand)
End Sub
```

Table 6-2: User Interface of Example Project

OpenFile is a user procedure that utilizes IDL’s DIALOG_PICKFILE function to enable the user to select a file for display within the **IDLDrawWidget**.

Cleaning Up and Exiting

Like the first example, this program exits when the user clicks “Exit”. An additional call has been made to **DestroyDrawWidget**. This isn’t necessary when exiting because the windowing system will destroy the widget. If you want to change the **GraphicsLevel** property of the **IDLDrawWidget** during program execution use this method.

IDL

```
PRO OpenFile, TLB
        WIDGET_CONTROL, TLB, GET_UVALUE = ptr
        PathName = DIALOG_PICKFILE(TITLE = $
                'Select a JPEG file', FILTER = '*.jpg')
        IF (PathName NE '') THEN BEGIN
                DEVICE, DECOMPOSED = 0
                READ_JPEG, PathName, Data, ColorTable
                (*(*ptr).OriginalArrayPTR) = Data
                (*(*ptr).OrigColorMapPTR) = ColorTable
                TVLCT, (*(*ptr).OrigColorMapPTR)
                TV, (*(*ptr).OriginalArrayPTR)
        ENDIF ELSE BEGIN
                Result = DIALOG_MESSAGE('No JPEG file selected', /ERROR)
        ENDELSE
END
```

Table 6-3: The Open File Procedure

Advanced Examples

Each of the following examples builds on the concepts that you've already learned in this chapter.

The user interface and projects for each of the examples have been created and can be found in the distribution in the `examples\doc\ActiveX\`*`project`* directory where *project* is the name of the example. These examples assume that you are already familiar with the following concepts:

- Creating a new project in Visual Basic;
- Adding the **IDLDrawWidget** control to the VB control toolbar;
- Drawing the **IDLDrawWidget** on your form;
- Initializing IDL with **InitIDL***;*
- Creating the draw widget with **CreateDrawWidget***;*
- Executing commands with **ExecuteStr***;*
- Using IDL `.pro` code to respond to auto-events within the **IDLDrawWidget**;
- Setting properties for the **IDLDrawWidget** objects.

These examples demonstrate the following:

- Copying and Printing IDL Graphics
- XLoadCT Functionality Using Visual Basic
- XPalette Functionality Using Visual Basic
- Integrating Object Graphics Using Visual Basic
- Sharing a Grid Control Array with IDL
- Handling Events within Visual Basic

Copying and Printing IDL Graphics

The *VBCopyPrint* example demonstrates how to use either the Windows clipboard or object graphics to print the contents of an **IDLDrawWidget** window.

This example illustrates the following concepts:

- Opening an existing project in Visual Basic;
- Copying an IDL graphic to the Windows clipboard using the **CopyWindow** method;
- Executing IDL user routines;
- Printing an IDL graphic.

Opening the VBCopyPrint project

Select “Existing” from the Visual Basic New Project dialog. In the IDL distribution, change to the `examples\docs\ActiveX\VBCopyPrint` directory, and open the project **VBCopyPrint.vbp**, as shown in the following figure.

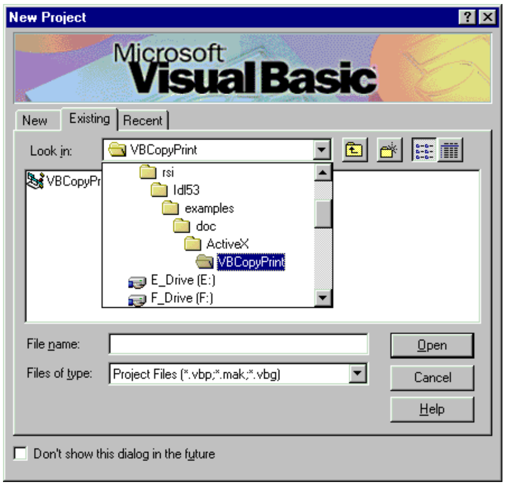


Figure 6-5: Opening the VBCopyPrint project

Running the VBCopyPrint Example

Select “Start” from the Run menu to run the example. You should see the graphic shown in the following figure.

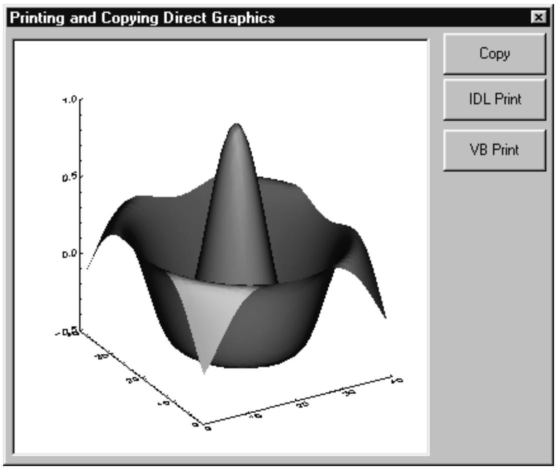


Figure 6-6: VBCopyPrint example

Copying IDL Graphic to the clipboard

To copy the graphic, click on “Copy”. The code for “Copy” uses the **CopyWindow** method to copy the contents of the graphic to the Windows clipboard as shown in line 6 of the following table.

Visual Basic	
	```
	  Private Sub cmdCopy_Click()
	1   'Copy the direct graphics window to the clipboard
	2   Screen.MousePointer = vbHourglass
	3   'Erase anything currently on the clipboard
	4   Clipboard.Clear
	5   'Copy the draw widget to the clipboard
	6   IDLDrawWidget1.CopyWindow
	7   Screen.MousePointer = vbDefault
	8   MsgBox "Window copied to clipboard."
	  End Sub
	```

Table 6-4: Copy button Source Code

Printing the IDL Graphic using IDL Object Graphics

To print the graphic using IDL, click on “IDL Print”. The “IDL Print” button uses IDL’s object graphics to print the contents of the window by creating an image object and sending the image to a printer object through a user routine **VBPrintWindow**.

IDL

```
PRO VBPrintWindow, DrawId
                  .
                  .
                  .
  ;Get the window index of the drawable to be printed
  WIDGET_CONTROL, DrawId, Get_Value=Index
                  .
                  .
                  .
  ;Create a Printer object and draw the graphic to it
  oPrinter = OBJ_NEW ('IDLgrPrinter')

  ;Display a print dialog box
  Result = DIALOG_PRINTERSETUP(oPrinter)
                  .
                  .
                  .
  oPrinter->Draw, oView
                  .
                  .
                  .
END ;VBPrintWindow
```

Table 6-5: IDL VBPrintWindow Code

Executing IDL user routines with Visual Basic

The **VBCopyPrint** example executes a user routine, written in IDL, to support the printing of the **IDLDrawWidget** window. This is done with the **ExecuteStr** method, as shown in line 4 below, by passing a string of the routine name along with the ID of the **IDLDrawWidget**.

Visual Basic

```
   Private Sub cmdPrintIDL_Click()
1    'Print the current drawable widget's window contents
2    'using IDL object graphics
3    Screen.MousePointer = vbHourglass
4    IDLDrawWidget1.ExecuteStr "VBPrintWindow," &
5      Str$(IDLDrawWidget1.DrawId)
6    Screen.MousePointer = vbDefault
     MsgBox "Window sent to printer."
   End Sub
```

Table 6-6: Print Button Source Code

Printing the IDL Graphic Using Visual Basic

The **VBPrint** command uses the Windows clipboard and Visual Basic printer support to print the IDL Graphic, as shown in the following table.

Visual Basic

```
Private Sub cmdPrintVB_Click()
  CommonDialog1.CancelError = True
    On Error GoTo ErrHandler
    CommonDialog1.ShowPrinter
'-- Copy the window's contents to the clipboard
    'Erase anything currently on the clipboard
    Clipboard.Clear
    IDLDrawWidget1.CopyWindow
  '-- Send the picture located on the clipboard,
  'to the printer
    Printer.PaintPicture Clipboard.GetData, 0, 0
    Printer.EndDoc  'Send it to the printer
Exit Sub
ErrHandler:

    Exit Sub
End Sub
```

Table 6-7: VBPrint Command

XLoadCT Functionality Using Visual Basic

The **VBLoadCT** example duplicates the XLOADCT functionality using a VB interface. The `VBLoadCT.pro` source code is a functional duplicate of XLOADCT with procedure calls replacing the `xloadct_event` procedure as well as IDL widgets being replaced by VB controls. See the following figure for more information. In addition, this example extends XLOADCT by adding the following features:

- Options menu by clicking the right mouse button on a color;
- Use of IDL syntax to create separate functions for red, blue and green;
- Ability to save user created color tables.

This example illustrates the following concepts:

- Modifying existing IDL library code for use with the **IDLDrawWidget***;*
- IDL to Visual Basic color table conversion.

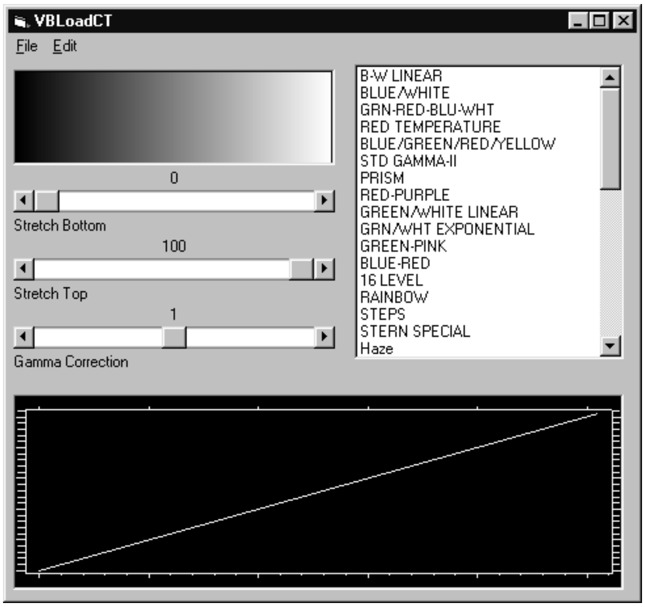


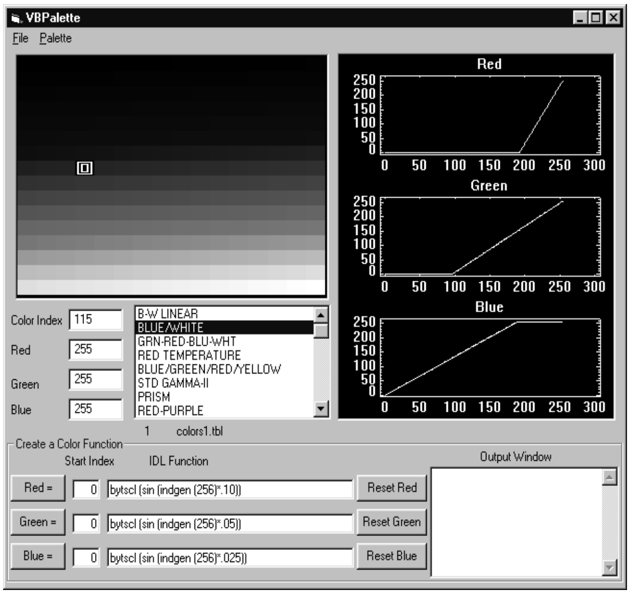
Figure 6-7: VBLoadCT example

XPalette Functionality Using Visual Basic

Like **VBLoadCT**, **VBPalette** demonstrates how to duplicate IDL tool functionality using a Visual Basic interface. The `VBPalette.pro` file is a functional duplicate of the **XPalette** source with the event procedure and IDL widgets replaced with auto-event procedures and VB controls.

This example illustrates the following concepts:

- Modifying existing IDL library code for use with the **IDLDrawWidget**;
- Converting an IDL event procedure to the **IDLDrawWidget** auto-event procedures.



Color Index 115
Red 255
Green 255
Blue 255

B-W LINEAR
BLUE/WHITE
GRN-RED-BLU-WHT
RED TEMPERATURE
BLUE/GREEN/RED/YELLOW
STD GAMMA-II
PRISM
RED-PURPLE

1 colors1.tbl

Start Index	IDL Function
0	bytscl (sin (indgen (256)*.10))
0	bytscl (sin (indgen (256)*.05))
0	bytscl (sin (indgen (256)*.025))

Figure 6-8: VBPalette Example

Integrating Object Graphics Using Visual Basic

Most of the examples covered to this point have used IDL's direct graphics sub-system to demonstrate using the **IDLDrawWidget** control. The **IDLDrawWidget** can also use IDL's object graphics sub-system by changing the **IDLDrawWidget**.**GraphicsLevel** property as demonstrated with the **VBObjGraph** example in the following figure.

This example illustrates the following concepts:

- Setting the **GraphicsLevel** property to create an object graphics window;
- Translating a graphics object using VB controls.
- Using **IDLDrawWidget** auto-events.

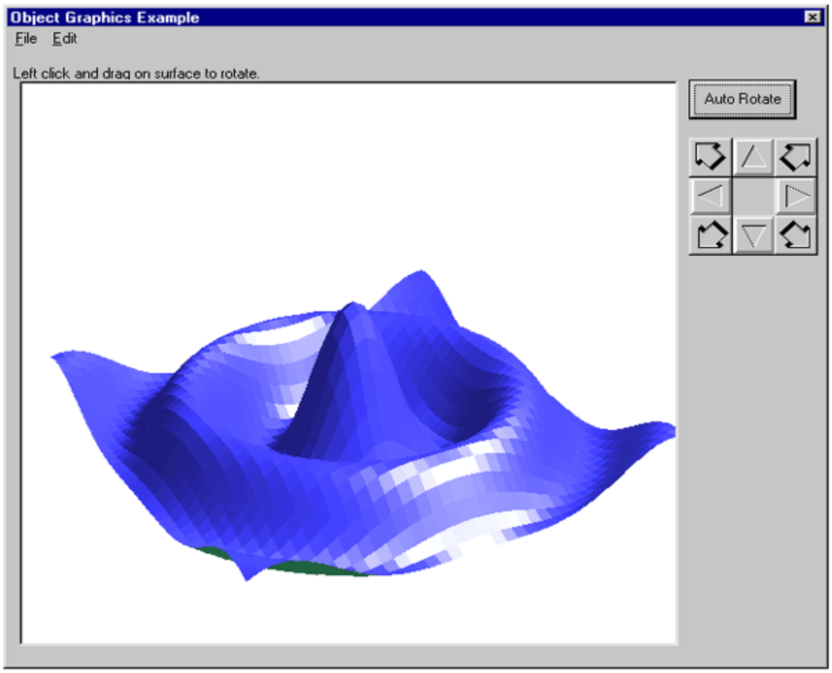


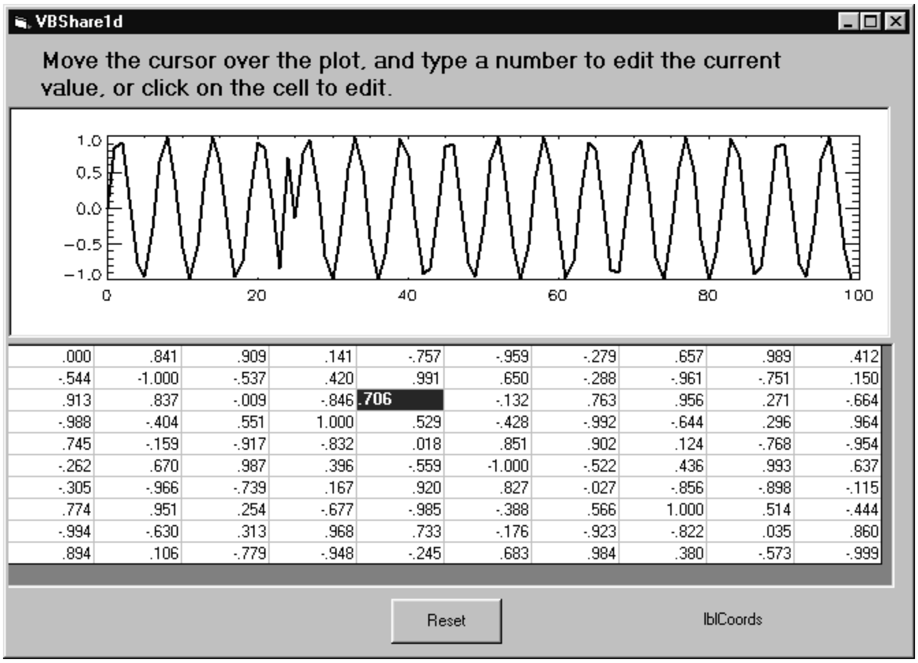
Figure 6-9: VBObjGraph example

Sharing a Grid Control Array with IDL

VBShare1D demonstrates sharing one dimensional data between Visual Basic and IDL using the **SetNamedArray** method of the **IDLDrawWidget** object. The data is presented to the user in a Visual Basic grid control enabling the user to edit the data and see the results in real time. See the following figure:

This example illustrates the following concepts:

- Shows how to process mouse events within VB to get the data coordinates of an IDL plot.
- Demonstrates how to convert (x,y) VB coordinates into IDL data coordinates, to give the cursor location in data values relative to the current plot.
- Demonstrates how to use a VB grid control to edit data values that are reflected in the IDL plot after each keystroke.



Move the cursor over the plot, and type a number to edit the current value, or click on the cell to edit.

.000	.841	.909	.141	-.757	-.959	-.279	.657	.989	.412
-.544	-1.000	-.537	.420	.991	.650	-.288	-.961	-.751	.150
.913	.837	-.009	-.846	.706	-.132	.763	.956	.271	-.664
-.988	-.404	.551	1.000	.529	-.428	-.992	-.644	.296	.964
.745	-.159	-.917	-.832	.018	.851	.902	.124	-.768	-.954
-.262	.670	.987	.396	-.559	-1.000	-.522	.436	.993	.637
-.305	-.966	-.739	.167	.920	.827	-.027	-.856	-.898	-.115
.774	.951	.254	-.677	-.985	-.388	.566	1.000	.514	-.444
-.994	-.630	.313	.968	.733	-.176	-.923	-.822	.035	.860
.894	.106	-.779	-.948	-.245	.683	.984	.380	-.573	-.999

Figure 6-10: VBShare1D

Handling Events within Visual Basic

The *VBPaint* example uses direct graphics to create a simple drawing program. A direct graphics window is used to respond to events within VB. Each click event will get the (x,y) location within the window, and modify the color of the current pixel in the image. See the following figure:

This example illustrates the following concepts:

- Converting from a VB pixel coordinate system to the IDL coordinate system;
- Converting a VB color representation (long) into an IDL color representation (RGB);
- Modifying an IDL RGB color table item with a color chosen/created from VB and the Window's common color dialog;
- Processing mouse events within VB to draw into an IDL window.

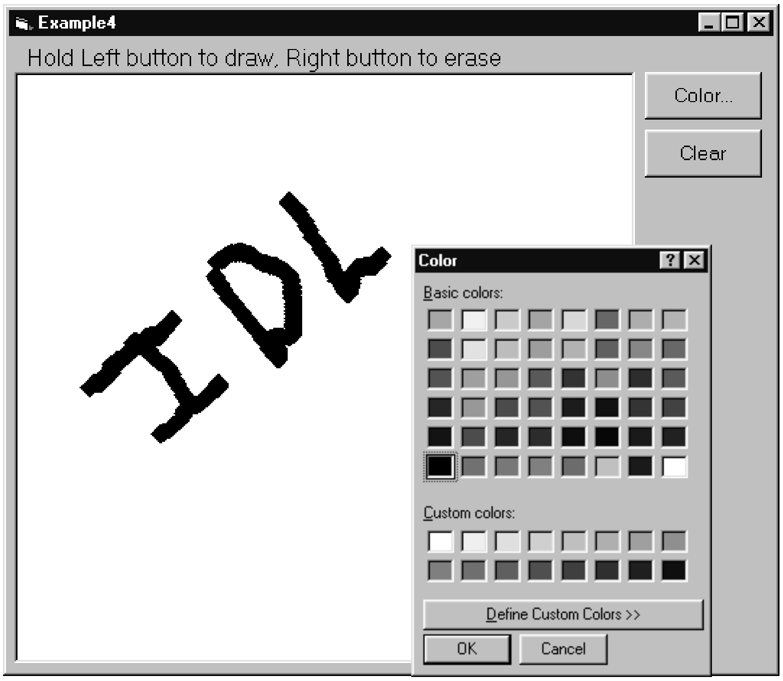
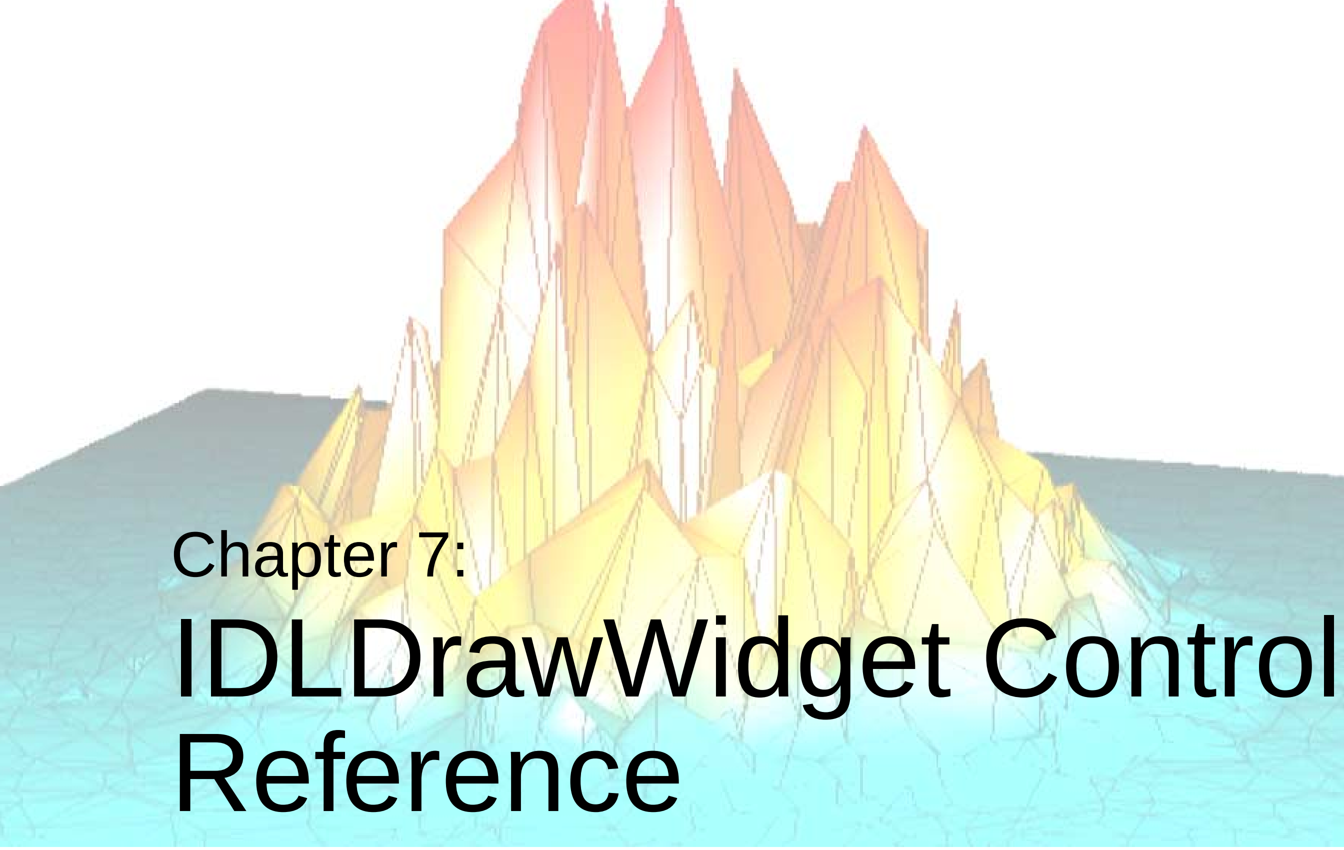


Figure 6-11: VBPaint example

Distributing Your ActiveX Application

For information on how to distribute an application developed with the IDL ActiveX control, see Chapter 20, “Creating IDL Projects” in the *Building IDL Applications* manual.



Chapter 7:

IDLDrawWidget Control Reference

This chapter describes the following topics:

IDLDrawWidget

The **IDLDrawWidget** is an ActiveX control that provides an easy mechanism for integrating IDL with Microsoft Windows applications written in C, C++, Visual Basic, Fortran, Delphi, etc. Methods and properties of the **IDLDrawWidget** provide the interface between IDL and an external application. The rest of this section describes the following for the **IDLDrawWidget**:

- Methods
- Do Methods (Runtime Only)
- Properties
- Read Only Properties
- Auto Event Properties
- Events

Methods

In ActiveX terminology, methods are special statements that execute on behalf of an object in a program. For example, the **ExecuteStr** method can be used to execute an IDL statement, function, or procedure when the user clicks on a button in a Visual Basic program. The syntax of a method statement is:

```
object.method value
```

where

- *Object* is the name of an object you want to control, for example an **IDLDrawWidget**.
- *Method* is the name of the method you want to execute.
- *Value* is an optional parameter used by the method. The various methods to the **IDLDrawWidget** may require zero, one, or multiple parameters.

Note

When a method returns a BOOL, the value TRUE is equal to 1 and FALSE is equal to 0.

CopyNamedArray

This method copies an IDL array to an OLE Variant array.

Parameters

BSTR: The name of the array variable that you wish to copy.

Returns

VARIANT: Reference to the array.

Remarks

This function returns an array reference that is local to the calling function. Attempting to use this array outside the calling function could result in runtime errors.

CopyWindow

This method copies the contents of the **IDLDrawWidget** window to the Windows clipboard.

Parameters

None.

Returns

BOOL: TRUE if successful.

CreateDrawWidget

This method creates an **IDLDrawWidget** in an ActiveX control frame. When you drag and drop the **IDLDrawWidget**, you are creating the frame that will contain the actual draw widget. Drawing operations to the control cannot be made until this method is called.

Parameters

None.

Returns

LONG: The widget ID of the created draw widget or -1 in the event of an error.

DestroyDrawWidget

This method destroys the **IDLDrawWidget**, but not the ActiveX control frame.

Parameters

None.

Returns

None.

DoExit

This method exits the ActiveX control and frees any resources in use by IDL.

After all IDL ActiveX control use is complete, but before the EDE application exits, you must call **DoExit** to allow the ActiveX control to shutdown IDL gracefully and free any resources in use.

Parameters

None.

Returns

None.

Remarks

In spite of the name, **DoExit** is not one of the IDL ActiveX control auto events. Like InitIDL, **DoExit** should be called once and only when you are exiting the EDE application.

Warning

Once **DoExit** is called, you are not allowed to call methods or set properties within the IDL ActiveX control from the currently running EDE application, regardless of which **IDLDrawWidget** the method was called on. Attempting to do so will result in a runtime error subsequently causing the EDE application to crash.

ExecuteStr

This method passes a string to IDL which IDL then executes.

Parameters

BSTR: A string containing the command that IDL will execute.

Returns

LONG: 0 if successful or the IDL error code if it fails.

Remarks

Most IDL commands that are executed with **ExecuteStr** run in the main level.

GetNamedData

This method returns the IDL data value associated with the named variable.

Parameters

BSTR: A string containing the name of an IDL variable.

Returns

VARIANT: Returns the value of the requested data. The type will be EMPTY if the IDL variable doesn't exist.

Remarks

The following table lists the supported IDL data types and the corresponding VARIANT data types.

IDL Type	Variant Type
IDL_TYP_BYTE	VT_UI1
IDL_TYP_INT	VT_I2
IDL_TYP_LONG	VT_I4
IDL_TYP_FLOAT	VT_R4
IDL_TYP_DOUBLE	VT_R8
IDL_TYP_STRING	VT_BSTR

Table 7-1: Supported IDL data types and the corresponding VARIANT data types

InitIDL

This method initializes IDL. IDL only needs to be initialized once for each instance of the ActiveX control.

Parameters

LONG: **InitIDL** is called with the **hWnd** of the main window for the container application. If this value is null, the ActiveX control uses the hWnd of the ActiveX control frame.

Returns

LONG: Long value indicating status of IDL

Value	Meaning
1	Successful
0	Failure
-1	IDL ActiveX control is not licensed
-2	IDL is unlicensed (demo)

Table 7-2: Status of IDL

If your application contains more than a single **IDLDrawWidget** (e.g. **IDLDrawWidget1** and **IDLDrawWidget2**) the **InitIDL** method should only be called on one of the objects, not both.

The **IDL ActiveX** control relies on IDL and must, at a minimum, have an IDL runtime distribution to operate successfully. The **IdlPath** property can be set so the control can find a valid IDL distribution (the `idl32.dll`). If a valid distribution is not found in either the path as set in the **IdlPath** property or the current directory, a dialog will be displayed giving the user the opportunity to specify the location of his IDL distribution. This behavior may be overridden at runtime by locating and specifying the path to the IDL distribution prior to calling either the **InitIDL** or **SetOutputWnd** methods.

InitIDLEx

This method initializes IDL. It is identical to the InitIDL method except that it has an additional parameter, **Flags**, allowing initialization flags to be passed on to IDL. See the description of the "InitIDL" on page 122 for details on the return value.

Parameters

LONG: **InitIDL** is called with the **hWnd** of the main window for the container application. If this value is null, the ActiveX control uses the **hWnd** of the ActiveX control frame.

LONG: **Flags**. A bitmask used to specify initialization options. The allowed bit values are:

Flag	Meaning
IDL_INIT_RUNTIME	Setting this bit causes IDL to check out a runtime license instead of the normal license. In Visual C++ applications, the define constant IDL_INIT_RUNTIME exported in `export.h` can be used. For Visual Basic applications use the actual value of this constant, IDL_INIT_RUNTIME=4, since the defined constant is not available.

Table 7-3: InitIDLEx Flags.

Returns

LONG: Long value indicating status of IDL. See the description of the return value under "InitIDL" on page 122 for details.

Print

This method prints the contents of the ActiveX control to the current default printer for both Direct and Object Graphics windows. The Print method will print the contents of a Direct Graphics window at screen resolution (72-96 dpi). For information about controlling print resolution of an object graphics window, see the BufferId property.

Note

In order to print the contents of an Object Graphics window, you must associate the IDL graphics tree (an IDLgrView object) with the IDLgrWindow object used by the underlying draw widget. Do this by setting the GRAPHICS_TREE property of the IDLgrWindow object to the IDLgrView object:

```
;Retrieve the window object associated with the draw widget.
IDLDrawWidget::ExecuteStr("Widget_Control, IDLDrawWidget, $
   Get_Value =oWindow");
;Set the Graphics_Tree property to the view object.
IDLDrawWidget::ExecuteStr("oWindow->SetProperty, $
   Graphics_Tree = oView");
```

Parameters

XOffset: The X offset to print the graphic in 0.01 of a millimeter.

YOffset: The Y offset to print the graphic in 0.01 of a millimeter.

Width: The desired width of the printed graphic in 0.01 of a millimeter.

Height: The desired height of the printed graphic in 0.01 of a millimeter.

The X offset plus the width should be less than or equal to the width of a single page. The Y offset plus the height should be less than or equal to the height of a single page. The origin of the offset 0,0 is in the upper left corner of a page. If these values are set to 0, the ActiveX control will print a graphic in the upper left corner of the page with the size of the graphic approximating the size of the image on the screen.

Returns

BOOL: TRUE if printing succeeded.

RegisterForEvents

This method causes **IDLDrawWidget** to pass the specified events to the application. These events only apply if the user hasn't set the corresponding auto event property.

Parameters

LONG: Flags that indicate which events you wish to forward to your application. Values can be combined if multiple events are desired.

Value	Meaning
0	Stop forwarding all events
1	Forward mouse move events
2	Forward mouse button events
4	Forward view scrolled events
8	Forward expose events

Table 7-4: Forwarding Events

Note

Motion events *may* be generated continuously in response to certain operations in IDL. As a result, if you forward mouse move events, your event handler should check the reported position of the mouse to determine whether it has in fact moved before doing extensive processing.

Returns

BOOL: TRUE if successful.

SetNamedArray

This method creates a named IDL array with the specified data. The data pointer is shared with IDL and the EDE application. Thus, changes in either IDL or the EDE will be reflected in both.

Parameters

BSTR: Name of array variable to create in IDL.

VARIANT: Array data to be shared with IDL.

BOOL: True if IDL should free a shared array when IDL releases its reference, false if not.

Returns

WORD: 1 if successful, 0 if set failed.

Remarks

Because **SetNamedArray** creates an array whose data is shared between IDL and the EDE application, IDL constructs that could change the type and/or dimensionality of the array must be avoided, as these constructs could have the side effect of creating a new array in IDL and thus breaking the shared link.

The array parameter of **SetNamedArray** must have a lifetime beyond the calling function. Thus, in Visual Basic, it is recommended that the array be declared as global in scope to prevent runtime errors from occurring.

Note

In order to allow data to be shared between IDL and the external environment, the lock count on the underlying array is incremented. Some external environments, notably later versions of Delphi, do not allow array locking to extend beyond a single method call and will signal an error when **SetNamedArray()** returns. If this occurs, the data cannot be shared between IDL and the external environment using

SetNamedArray(). Use the **SetNamedData()** method to insert a copy of the array into IDL.

The following table lists the accepted variant types and the corresponding IDL types.

Variant Types	IDL Types
VT_UI1 - unsigned char	IDL_TYP_BYTE
VT_I1 - signed char	IDL_TYP_BYTE
VT_I2 - signed short	IDL_TYP_INT
VT_I4 - signed long	IDL_TYP_LONG
VT_R4 - float	IDL_TYP_FLOAT
VT_R8 - double	IDL_TYP_DOUBLE

Table 7-5: Accepted Variant Types and the Corresponding IDL Types

SetNamedData

This method creates an IDL variable with the specified name and value. Both the EDE and IDL maintain their own copy of the data. **SetNamedData** can also be used to change the value of an existing IDL variable.

Parameters

BSTR: Name of the variable to create in IDL.

VARIANT: Data to be copied in IDL.

Returns

WORD 1 if successful.

SetOutputWnd

This method sends output from IDL to the specified window.

Parameters

HWND: The **hWnd** of the edit control that will receive the output.

Returns

None.

Note

SetOutputWnd is the only method that can be called prior to a call to **InitIDL**.

VariableExists

This method determines if a specified variable is defined in IDL.

Parameters

BSTR: Name of variable to check.

Returns

BOOL:TRUE if variable is defined in IDL at the main level. False if the variable is not defined.

Do Methods (Runtime Only)

Do Methods are methods that execute auto event procedures. Calling these methods is helpful in simulating user interaction with a draw widget by forcing an auto event to be called.

DoButtonPress

This method calls the IDL procedure specified in the **OnButtonPress** property.

Parameters

None.

Returns

None.

DoButtonRelease

This method calls the IDL procedure specified in the **OnButtonRelease** property.

Parameters

None.

Returns

None.

DoExpose

This method calls the IDL procedure specified in the **OnExpose** property.

Parameters

None.

Returns

None.

DoMotion

This method calls the IDL procedure specified in the **OnMotion** property.

Parameters

None.

Returns

None.

Properties

Properties are used to specify the various attributes of an **IDLDrawWidget**, such as its color, width and height. Most properties may be set at design time by configuring the properties sheet in Visual Basic, or at runtime by executing statements in the program code.

The syntax for setting a property in the code is:

```
object.property = value
```

where

- *Object* is the name of the object you want to change, e.g. **IDLDrawWidgetn** where *n* is the number Visual Basic assigned to the **IDLDrawWidget**.
- *Property* is the characteristic you want to change.
- *Value* is the new property setting.

Note

All properties relating to window size and/or position are in pixel units unless otherwise indicated.

BackColor

This property specifies the background color of the IDL widget. **BackColor** may be specified at design time or runtime.

BaseName

This property names a variable that IDL will use for the pseudo base. If this property is set, the **IDLDrawWidget** will create an IDL variable with this name that contains the ID of the base widget. Because the base widget is a pseudo base, you should not destroy it. The **BaseName** property can be set at design time or at runtime prior to a call to **CreateDrawWidget**.

Default=**IDLDrawWidgetBase**

BufferId

The BufferId controls the type of print output you receive when printing with an Object Graphics window (when the GraphicsLevel property is set to 2).

1. A value of -1 will cause the graphics to print using vector output. This format is suitable for line graphs and mesh surfaces.
2. A value of 0 will cause the graphics to print at roughly two times the screen resolution. This format is suitable for shaded surfaces or vertex colored mesh surfaces. This is the default.
3. A value greater then 0 will be construed a s an IDLgrBuffer object reference whose data will be used for printing. This format allows the programmer to control the resolution of the output of the image.

For more information, see "IDLgrBuffer" in the *IDL Reference Guide* manual.

Note

You must set the GRAPHICS_TREE property of the IDLgrWindow object for these print options to work.

DrawWidgetName

Returns or sets a variable that IDL will use for the draw widget. If this property is set, the **IDLDrawWidget** will create an IDL variable with this name that contains the ID of the draw widget. The **DrawWidgetName** property can be set at design time, or at runtime prior to a call to **CreateDrawWidget**.

Default=**IDLDrawWidget**

Enabled

Returns or sets a value that determines whether a form or control can respond to user-generated events such as mouse events.

Default=TRUE

GraphicsLevel (Runtime/Design time)

This property specifies the graphics level of the draw widget. Legal values are 1 or 2. If you set the **GraphicsLevel = 1** and call the **CreateDrawWidget** method, the procedure will create an IDL direct graphics window. **GraphicsLevel = 2** results in an IDL object graphics window. The **GraphicsLevel** property can be set at design time or at runtime prior to a call to **CreateDrawWidget**.

Default=1

IdlPath

This property specifies the fully qualified path to the IDL32.DLL. The **IdlPath** property can be set at design time or at runtime prior to a call to **InitIDL** or **SetOutputWnd**.

Default=NULL

Renderer

This property specifies either the software or hardware renderer for object graphics windows is to be used. It has no effect if the GraphicsLevel property is set to 1. Valid values are:

- 0 = Platform native OpenGL
- 1 = IDL's software implementation

By default, the setting in your IDL preferences is used.

Retain (Runtime/Design time)

This property sets the retain mode of the IDLDrawWidget: 0, 1, or 2. The retain mode specifies how IDL should handle backing store for the draw widget. **Retain=0** specifies no backing store. **Retain=1** requests that the server or window system provide backing store. **Retain=2** specifies that IDL provide backing store directly. The **Retain** property can be set at design time or at runtime prior to a call to **CreateDrawWidget**.

Default=1

Visible (Runtime/Design time)

Shows or hides the IDLDrawWidget. When Visible is TRUE the IDLDrawWidget is shown, when FALSE the IDLDrawWidget is hidden. Hiding the IDLDrawWidget is useful when the control is used as an interface to IDL and no graphics are intended for display.

Default=TRUE

Xsize (Design time)

Virtual width of **IDLDrawWidget**. If this value is greater than the **Xviewport** value, scroll bars will be added.

Ysize (Design time)

Virtual height of **IDLDrawWidget**. If this value is greater than the **Yviewport** value, scroll bars will be added.

Read Only Properties

BaseId (Runtime)

Widget ID of the pseudo base. The **BaseId** property is not valid until a call to **CreateDrawWidget** has been made.

DrawId (Runtime)

Widget ID of the created draw widget. The **DrawId** property is not valid until a call to **CreateDrawWidget** has been made.

hWnd (Runtime)

Window handle of the ActiveX control. The **hWnd** property is not valid until a call to **CreateDrawWidget** has been made.

LastIdlError (Runtime)

A string that contains the last IDL error message. This string will not change if the ExecuteStr method is called and an error does not occur.

Scroll

True if the widget will contain scroll bars.

Default=FALSE

Xoffset

Set at design time when the control is dropped or moved. Represents the x offset of the draw widget within the parent application.

Xviewport

Set at design time when the control is dropped or moved. Represents the visible width of the draw widget. If scroll bars are present **Xviewport** will include the width of the scroll bars.

Yoffset

Set at design time when the control is dropped or moved. Represents the y offset of the draw widget within the parent application.

Yviewport

Set at design time when the control is dropped or moved. Represents the visible height of the draw widget. If scroll bars are present **Yviewport** will include the height of the scroll bars.

Auto Event Properties

Auto events are IDL procedures that are called automatically by the control in response to certain events.

OnButtonPress

An IDL procedure that will be called when a mouse button is pressed. The procedure must be in the form:

```
pro button_press, drawId, button, xPos, yPos
```

Default=NULL

OnButtonRelease

An IDL procedure that will be called when a mouse button is released. The procedure must be in the form:

```
pro button_release, drawId, button, xPos, yPos
```

Default=NULL

OnDblClick

An IDL procedure that will be called when a mouse button is double clicked within the draw widget. The procedure must be in the form:

```
pro button_dblclick, drawId, button, xPos, yPos
```

The following table describes each parameter of the syntax:

Parameter	Description
button	Describes which mouse button has been clicked. The valid values are: • 1 — Left mouse button. • 2 — Middle mouse button. • 4 — Right mouse button.

Table 7-6: OnDblClick Parameters

Parameter	Description
xPos	The horizontal position of the mouse when the button was clicked.
yPos	The vertical position of the mouse when the button was clicked.

Table 7-6: (Continued) OnDblClick Parameters (Continued)

Default=NULL

OnExpose

An IDL procedure that will be called when an expose message is received by the draw widget. The procedure must be in the form:

```
pro expose, drawId
```

Default=NULL

OnInit

An IDL procedure that will be called when a draw widget is initially created. The procedure must be in the form:

```
pro init, drawId, baseId
```

This auto event procedure is called once when the **CreateDrawWidget** method is invoked.

Default=NULL

OnMotion

An IDL procedure that will be called when the mouse is moved over the draw widget while a mouse button is pressed. The procedure must be in the form:

```
pro motion, drawId, button, xPos, yPos
```

Default=NULL

Note

Motion events *may* be generated continuously in response to certain operations in IDL. As a result, if you provide an event-handler for mouse motion events, your event handler should check the reported position of the mouse to determine whether it has in fact moved before doing extensive processing.

Events

Events are functions or procedures that can be handled by the EDE application on behalf of **IDLDrawWidget**. If an auto event property is set, its corresponding event will not be called; instead, the auto event procedure will be called. By disabling the auto-events, **IDLDrawWidget** can respond to the following standard Visual Basic events:

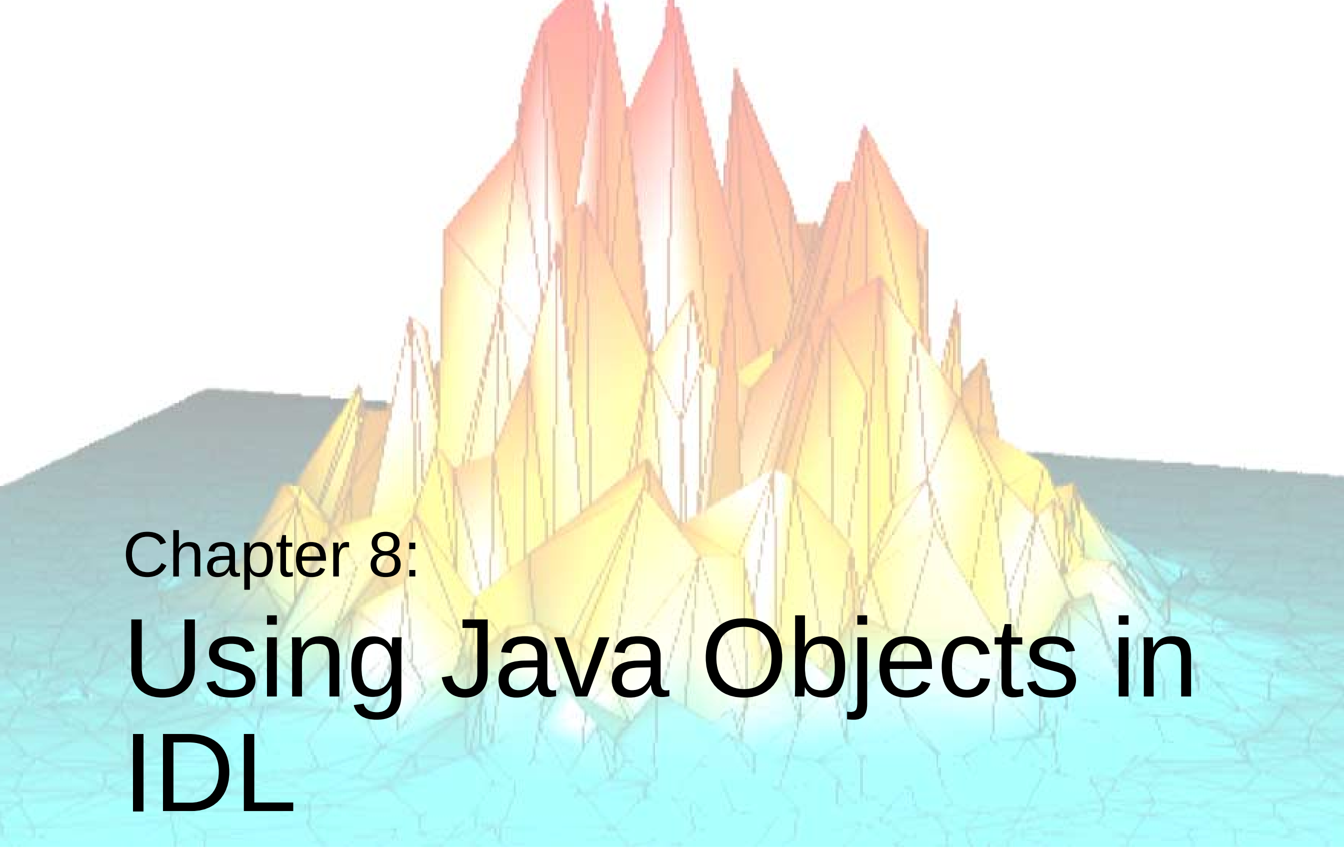
- MouseDown
- MouseMove
- MouseUp

OnViewScrolled

OnViewScrolled is an **IDLDrawWidget** event that notifies the container application when the graphics window has been scrolled. This event will only be sent when the **Scroll** property is TRUE.

Note

You must call **RegisterForEvents** passing the flags to indicate the events you want to process. Neglecting this step will send the events to IDL for processing.



Chapter 8: Using Java Objects in IDL

The following topics are covered in this chapter:

Overview

Java is an object-oriented programming language developed by Sun Microsystems that is commonly used for web development and other programming needs. It is beyond the scope of this chapter to describe Java in detail. Numerous third-party books and electronic resources are available. The Java website (http://java.sun.com) may be useful.

IDL 6.0 introduces the IDL-Java bridge, which allows you to access Java objects within IDL code. Java objects imported into IDL behave like normal IDL objects. See "Creating IDL-Java Objects" on page 153 for more information. The IDL-Java bridge allows the arrow operator (`->`) to be used to call the methods of these Java objects just as with other IDL objects, see "Method Calls on IDL-Java Objects" on page 155 for more information. The public data members of a Java object are accessed through GetProperty and SetProperty methods, see "Managing IDL-Java Object Properties" on page 157 for more information. These objects can also be destroyed with the OBJ_DESTROY routine, see "Destroying IDL-Java Objects" on page 159 for more information.

Note

IDL requires an evaluation or permanent IDL license to use this functionality. This functionality is not available in demo mode.

The bridge also provides IDL with access to exceptions created by the underlying Java object. This access is provided by the IDLJavaBridgeSession object, which is a Java object that maintains exceptions (errors) during a Java session, see "The IDLJavaBridgeSession Object" on page 161 for more information.

Note

Visual Java objects cannot be embedded into IDL widgets.

Currently, the IDL-Java bridge is supported on the Windows, Linux, Solaris, and Macintosh platforms supported in IDL. See "Requirements for this Release" in Chapter 1 of the *What's New in IDL 6.0* manual for more information on these platforms supported in IDL 6.0.

Java Terminology

You should become familiar with the following terms before trying to understand how IDL works with Java objects:

Java Virtual Machine (JVM) - A software execution engine for executing the byte codes in Java class files on a microprocessor.

Java Native Interface (JNI) - Standard programming interface for accessing Java native methods and embedding the JVM into native applications. For example, JNI may be used to call C/C++ functionality from Java or JNI can be used to call Java from C/C++ programs.

Java Invocation API - An API by which one may embed the Java Virtual Machine into your native application by linking the native application with the JVM shared library.

Java Reflection API - Provides a small, type-safe, and secure API that supports introspection about the classes and objects. The API can be used to:

- construct new class instances and new arrays
- access and modify fields of objects and classes
- invoke methods on objects and classes
- access and modify elements of arrays.

IDL-Java Bridge Architecture

The IDL-Java bridge uses the Java Native Interface (JNI), the reflection API, and the JVM to enable the connection between IDL and the underlying Java system.

The IDL OBJ_NEW function can be used to create a Java object. A Java-specific class token identifies the Java class used to create a Java proxy object. IDL parses this class name and creates the desired object within the underlying Java environment.

The Java-specific token is a case-insensitive form of the name of the Java class. Besides the token, the case-sensitive form of the name of the Java class is also provided because Java itself is case-sensitive while IDL is not. IDL uses the case-insensitive form to create the object definition while Java uses the case-sensitive form.

After creation, the object can then be used and manipulated just like any other IDL object. Method calls are the same as any other IDL object, but they are vectored off to an IDL Java system, which will call the appropriate Java method using JNI.

The OBJ_DESTROY procedure in IDL is used to destroy the object. This process releases the internal Java object and frees any resources associated with it.

Initializing the IDL-Java Bridge

The IDL-Java bridge must be configured before trying to create and use Java objects within IDL. The IDL program initializes the bridge when it first attempts to create an instance of IDLjavaObject. Initializing the bridge involves starting the Java Virtual Machine, creating any internal Java bridge objects (both C++ and Java) including the internal IDLJavaBridgeSession object. See "The IDLJavaBridgeSession Object" on page 161 for more information on the session object.

Configuring the Bridge

The `.idljavabrc` file on UNIX or `idljavabrc` on Windows contains the IDL-Java bridge configuration information. Even though the IDL installer attempts to create a valid working configuration file based on IDL location, the file should be verified before trying to create and use Java objects within IDL.

The IDL-Java bridge looks for the configuration file in the following order:

1. If the environment variable $IDLJAVAB_CONFIG is set, the file it indicates is used.

   Note

   This environment variable must include both the path AND the file name of the configuration file.

2. If the environment variable $IDLJAVAB_CONFIG is not set or the file indicated by that variable is not found in that location, the path specified in the $HOME environment variable is used to try to locate the configuration file.
3. If the file is not found in the path indicated by the $HOME environment variable, the <IDL_DEFAULT>`/external/objbridge/java` path is used to try to locate the configuration file.

The configuration file contains the following settings. With a text editor, open your configuration file to verify these settings are correct for your system.

- The `JVM Classpath` setting specifies additional locations for user classes. It must point to the location of any class files to be used by the bridge. On Windows, paths should be separated by semi-colons. On UNIX, colons should separate paths.

This path may contain folders that contain class files or specific jar files. It follows the same rules for specifying '-classpath' when running `java` or `javac`. You can also include the $CLASSPATH environment variable in the `JVM Classpath`:

```
JVM Classpath = $CLASSPATH:/home/johnd/myClasses.jar
```

which allows any class defined in the CLASSPATH environment variable to be used in the IDL-Java bridge.

On Windows, an example of a typical `JVM Classpath` setting is:

```
JVM Classpath = E:\myClasses.jar;$CLASSPATH
```

On UNIX, an example of a typical `JVM Classpath` setting is:

```
JVM Classpath = /home/johnd/myClasses.jar:$CLASSPATH
```

- The `JVM LibLocation` setting tells the IDL-Java bridge which JVM shared library within a given Java version to use. Various versions of Java ship with different types of JVM libraries. For example, Java 1.3 on Windows ships with a "classic" JVM, a "hotspot" JVM, and a "server" JVM. Other versions and platforms have different JVM types.

  On Windows, an example of a typical `JVM LibLocation` setting is:

  ```
  JVM LibLocation = E:\jdk1.3.1_02\jre\bin\hotspot
  ```

  On UNIX, an example of a typical `JVM LibLocation` setting is

  ```
  JVM LibLocation = /usr/java/j2re1.4.0_02/lib/sparc/client
  ```

Note

The preferred method for setting `JVM LibLocation` on Windows is via the configuration file or the IDLJAVAB_LIB_LOCATION environment variable. The preferred method on UNIX is via the $IDLJAVAB_LIB_LOCATION environment variable because UNIX requires this variable to be set in order to find Java shared libraries.

- The `JVM Option#` (where # is any whole number) setting allows you to send additional parameters to the Java Virtual machine upon initialization. These settings must be specified as string values. When these settings are encountered in the initialization, the options are added to the end of the options that the bridge sets by default.
- The `Log Location` setting indicates the directory where IDL-Java bridge log files will be created. The default location provided by the IDL installer is `/tmp` on Unix and `c:\temp` on Windows.

- The `Bridge Logging` setting indicates the type of bridge debug logging to be sent to a file called `jb_log<`*`pid`*`>.txt` (where `<`*`pid`*`>` is a process ID number) located in the directory specified by the `Log Location` setting.

  Acceptable values (from least verbose to most verbose) are `SEVERE`, `CONFIG`, `CONFIGFINE`. The default value is `SEVERE`, which specifies that bridge errors are logged. The `CONFIG` value indicates the configuration settings are also logged. The `CONFIGFINE` value is the same as `CONFIG`, but provides more detail.

  No log file is created if this setting is not specified.

The IDL-Java bridge usually only uses the configuration file once during an IDL session. The file is used when the first instance of the IDLjavaObject class is created in the session. If you edit the configuration file after the first instance is created, you must exit and restart IDL to update the IDL-Java bridge with the changes you made to the file.

IDL-Java Bridge Data Type Mapping

When data moves between IDL and a Java object, IDL automatically converts variable data types.

The following table maps how Java data types correlate to IDL data types.

Java Type (# bytes)	IDL Type	Notes
boolean (1)	Integer	True becomes 1, false becomes 0
byte (1)	Byte	
char (2)	Byte	The bridge handles Java UTF characters
short (2)	Integer	
int (4)	Long	
long (8)	Long64	
float (4)	Float	
double (8)	Double	
Java.lang.String	String	Java has the notion of a NULL string (the java.lang.String reference equals null) and the concept of an empty string. IDL makes no such differentiation, so both are identically converted.
Arrays of the above types	IDL array of the same dimensions (from 1 to 8 dimensions) and corresponding type.	

Table 8-1: Java to IDL Data Type Conversion

Java Type (# bytes)	IDL Type	Notes
Java.lang.Object (or array of java.lang.Object) and any subclass of java.lang.Object	IDL array of primitives or IDL array of IDLjavaObjects	In Java, everything is a subclass of Object. If the Java object is an array of primitives, an IDL array of the same dimensions and corresponding type (shown in this table) is created. IDL similarly converts arrays of primitives, arrays of strings, arrays of other Java objects to an IDL Java object of the same dimensions. If the Object is some single Java object, IDL creates an object reference of the IDLjavaObject class.
Null object	IDL Null object	

Table 8-1: Java to IDL Data Type Conversion (Continued)

The following table shows how data types are mapped from IDL to Java.

IDL Type	Java Type (# bytes)	Notes
Byte	byte (1)	IDL bytes range from 0 to 255, Java bytes are -128 to 127. IDL bytes converted to Java bytes will retain their binary representation but values greater than 127 will change. For example, BYTE(255) becomes a Java byte of -1. If BYTE is converted to wider Java value, the sign and value is preserved.
Integer	short (2)	
Unsigned integer	short (2)	IDL unsigned integers range from 0 to 65535, Java shorts are -32768 to 32767. IDL unsigned integers converted to Java shorts will retain their binary representation but values greater than 32768 will change. For example, UINT(65535) becomes a Java short of -1. If UINT is converted to wider Java value, the sign and value is preserved.
Long	int (4)	

Table 8-2: IDL to Java Data Type Conversion

IDL Type	Java Type (# bytes)	Notes
Unsigned long	int (4)	IDL unsigned longs range from 0 to 4294967295, Java ints are -2147483648 to 2147483647. IDL unsigned longs converted to Java ints will retain their binary representation but values greater than 2147483647 will change. For example, ULONG(4294967295) becomes a Java int of -1. If ULONG is converted to wider Java value, the sign and value is preserved.
Long64	long (8)	
Unsigned Long64	long (8)	IDL unsigned long64 range from 0 to 18446744073709551615, Java ints range from -9223372036854775808 to 9223372036854775807. IDL unsigned long64 converted to Java longs will retain their binary representation values greater than 9223372036854775807 will change. For example, ULONG64(18446744073709551615) becomes a Java long of -1.
Float	float (4)	
Double	double (8)	
String	Java.lang.String	
Arrays of the above types	Java array of the same dimensions and corresponding type	

Table 8-2: IDL to Java Data Type Conversion (Continued)

IDL Type	Java Type (# bytes)	Notes
IDLjavaObject	Object of corresponding Java class	
Arrays of objects	Java array of the same dimensions, consisting of corresponding Java proxy objects	Only objects of type IDLjavaObject are converted.
Null object	Java null	

Table 8-2: IDL to Java Data Type Conversion (Continued)

When calling a Java method or constructor from IDL, the data parameters are promoted as little as possible based on the signature of the given method. The following table shows how data types are promoted within Java relative to IDL.

Note

When strings and arrays are passed between IDL and Java, the array must be copied. Depending upon the size of the array, this copy may be time intensive. Care should be taken to minimize array copying.

IDL Type	Java Type (to order of desired promotion)	Notes
Byte	byte, char, short, int, long, float, double, boolean	
Integer	short, int, long, float, double, boolean	
Unsigned integer	short, int, long, float, double, boolean	
Long	int, long, float, double, boolean	
Unsigned Long	int, long, float, double, boolean	
Long64	long, float, double, boolean	
Unsigned Long64	long, float, double, boolean	

Table 8-3: Java Data Type Promotion Relative to IDL

IDL Type	Java Type (to order of desired promotion)	Notes
Float	float, double	
Double	double	
String	Java.lang.String	
IDLjavaObject	Java.lang.Object	

Table 8-3: Java Data Type Promotion Relative to IDL (Continued)

Creating IDL-Java Objects

As with all IDL objects, a Java object is created using the IDL OBJ_NEW function. Keying off the provided Java class name, the underlying implementation uses the IDL Java subsystem to call the constructor on the desired Java object. The following line of code demonstrates the basic syntax for calling OBJ_NEW to create a Java object within IDL:

```
oJava = OBJ_NEW(IDLjavaObject$JAVACLASSNAME, JavaClassName, $
   [Arg1, Arg2, ..., ArgN])
```

where *JAVACLASSNAME* is the class name token used by IDL to create the object, *JavaClassName* is the class name used by Java to initialize the object, and *Arg1* through *ArgN* are any data parameters required by the constructor. See "Java Class Names in IDL" for more information.

See the `hellojava.pro` file in the `external/objbridge/java/examples` directory of the IDL distribution for a simple example of an IDL-Java object creation.

Note

If you edit and recompile a Java class used by IDL during an IDL-Java bridge session, you must first exit and restart IDL before your modified Java class will be recognized by IDL.

The IDL-Java bridge also provides the ability to access static Java methods and data members. See "Java Static Access" on page 154 for more information.

Java Class Names in IDL

The underlying Java interpreter recognizes the Java class name including all objects contained within the Java interpreter's class path.

To identify a proper Java object, the fully-qualified package name should be used when creating the IDL class name. For example, a class of type String would be referred to as `java.lang.String`.

In the IDL class name, the Java class separator ('.') should be replaced with an underscore ('_'). If a Java class of type String were created, the following IDL OBJ_NEW call would be used:

```
oJString = OBJ_NEW('IDLJavaObject$JAVA_LANG_STRING',$
   'java.lang.String', 'My String')
```

The class name is provided twice because IDL is case-insensitive whereas Java is case-sensitive, see "IDL-Java Bridge Architecture" on page 143 for more information.

Note

IDL objects use method names (INIT and CLEANUP) to identify and call object lifecycle methods. As such, these method names should be considered reserved. If an underlying Java object implements a method using either INIT or CLEANUP, those methods will be overridden by the IDL methods and not accessible from IDL. In Java, you can wrap these methods with different named methods to work around this limitation.

Java Static Access

In Java, a program can call a static method or access static data members on a Java class without first having to create the object.

IDL contains a special wrapper object type for calling static methods. This IDL object wrapper references the underlying Java class, allowing the object to call static methods on the class or allowing the object to use the Get/Set Property calls to access static data members. The following line of code demonstrates the basic syntax for calling OBJ_NEW to create a static proxy within IDL:

```
oJava = OBJ_NEW(IDLjavaObject$Static$JAVACLASSNAME, JavaClassName)
```

where *JAVACLASNAME* is the class name token used by IDL to create the object and *JavaClassName* is the class name used by Java to initialize the object. See "Java Class Names in IDL" on page 153 for more information.

A special static object would not need to be created to call an instantiated `IDLJavaObject` with static methods:

```
oNotStatic = OBJ_NEW('IDLjavaObject$JAVACLASSNAME', $
   'JavaClassName')
oNotStatic -> aStaticMethod ; this is OK
```

See the `javaprops.pro` file in the `external/objbridge/java/examples` directory of the IDL distribution for an example of working with static data members.

Note

All restrictions on creating Java objects apply to this static object.

Method Calls on IDL-Java Objects

When a method is called on a Java-based IDL object, the method name and arguments are passed to the IDL-Java subsystem and the Java Reflection API to construct and invoke the method call on the underlying object.

IDL handles conversion between IDL and Java data types. Any results are returned in IDL variables of the appropriate type.

As with all IDL objects, the general syntax in IDL for an underlying Java method that returns a value (known as a function method in IDL) is:

```
result = ObjRef -> Method([Arguments])
```

and the general syntax in IDL for an underlying Java method that does not return a value, a void method, (known as a procedure method in IDL) is:

```
ObjRef -> Method[, Arguments]
```

where `ObjRef` is an object reference to an instance of a dynamic subclass of the IDLjavaObject class.

Note

Besides other Java based objects, the value of an argument may be an IDL primitive type, an IDLjavaObject, or an IDL primitive type array. No complex types (structures, pointers, etc.) are supported as parameters to method calls.

What Happens When a Method Call is Made?

When a method is called on an instance of IDLjavaObject, IDL uses the method name and arguments to construct the appropriate method calls for the underlying Java object.

From the point of view of an IDL user issuing method calls on an instance of IDLjavaObject, this process is completely transparent. IDL handles the translation when the IDL user calls the Java object’s method.

Due to case-sensitivity incompatibilities between IDL and Java, Java’s ability to overload methods, and the fact that Java might promote certain data types, the Java bridge uses an algorithm to match the IDL method name and parameters to the corresponding Java object method.

Before the algorithm starts, IDL provides a case-insensitive <METHODNAME> and a reference to the Java object. For a given object and its parent classes, the Java bridge obtains a list of all the public method names, including static methods. This algorithm performs the following steps:

1. If the Java class has one method name matching the IDL <METHODNAME> (except for case insensitivity), this Java method name is used. At this point, signatures and overloaded functions are not taken into account.
2. If the Java class has several method names that differ only in case and one is all uppercase, the uppercase name is used. Otherwise, the IDL-Java bridge issues an error that it has no method named <METHODNAME>.
3. Once the method name has been determined, a promotion algorithm then matches the Java data parameters as closely as possible with the IDL parameters. Minimum data promotion from IDL to Java is preferred and only widening promotion is allowed. If no match is found, an error is issued.

Data Type Conversions

IDL and Java use different data types. IDL's dynamic type conversion facilities handle all conversion of data types between IDL and the Java system. The data type mappings are described in "IDL-Java Bridge Data Type Mapping" on page 147.

For example, if the Java object has a method that requires a value of type `int` as an input argument, IDL would supply the value as an IDL Long. For any other IDL data type, IDL would first convert the value to an IDL Long using its normal data type conversion mechanism before passing the value to the Java object as an `int`.

Managing IDL-Java Object Properties

Property names and arguments are also passed to the IDL Java subsystem and are used in conjunction with the Java Reflection API to construct and access public data members on the underlying object. These public data members (known as properties in IDL) are identified through arguments to the GetProperty and SetProperty methods. See "Getting and Setting Properties" on page 158 for more information.

Note

Only public data members may be accessed.

Due to case-sensitivity incompatibilities between IDL and Java and the fact that Java might promote certain data types, the Java bridge uses an algorithm to match the IDL properties name to the corresponding Java object data members.

Before the algorithm starts, IDL provides a case-insensitive <PROPERTYNAME> and a reference to the Java object. For the given object and its parent classes, the Java bridge obtains a list of all the public data members including static members. This algorithm performs the following steps:

1. If the Java class has one data member name matching the IDL <PROPERTYNAME> (except for case insensitivity), this Java data member is used. At this point, data types are not yet taken into account; this algorithm only matches the data member names.
2. If the Java class has several member names that differ only in case, the data member name that exactly matches the IDL < PROPERTYNAME > (i.e. the one that is all caps) is called. Otherwise, the IDL-Java bridge issues an error that the class has no data members named < PROPERTYNAME >.
3. When setting a property with the SetProperty method, a promotion algorithm matches the provided IDL parameter with the Java data parameter as closely as possible. If the IDL value can be promoted to the same type as the data member, this data member is used. Otherwise, an error is issued.

   When retrieving a property with the GetProperty method, this step is skipped and the value is returned to IDL.

See the `allprops.pro` and `publicmembers.pro` files in the `external/objbridge/java/examples` directory of the IDL distribution for IDL routines that provide information about data members associated with given Java classes.

Getting and Setting Properties

The IDL-Java bridge follows the standard IDL property interface to support data member access on Java objects and classes.

To retrieve a property value from a Java object, use the following syntax:

```
ObjRef -> GetProperty, PROPERTY=variable
```

where `ObjRef` is an instance of IDLjavaObject that encapsulates the Java object, *PROPERTY* is the name of the Java object's data member (property), and *variable* is the name of an IDL variable that will contain the retrieved property value.

To retrieve multiple property values in a single statement supply multiple *PROPERTY=variable* pairs separated by commas.

To set a property value on a Java object, use the following syntax:

```
ObjRef -> SetProperty, Property=value
```

where `ObjRef` is an instance of IDLjavaObject that encapsulates the Java object, *PROPERTY* is the name of the Java object's data member, and *value* is value of the property to be set.

To set multiple property values in a single statement supply multiple *PROPERTY=value* pairs separated by commas.

Note

The provided *PROPERTY* must map directly to a data member name. Any name passed into either of the property routines is assumed to be a fully qualified Java property name. As such, the partial property name functionality provided by IDL is not valid with IDL Java based objects.

The *variable* or *value* part may be an IDL primitive type, an instance of IDLJavaObject, or an array of an IDL primitive type. See "IDL-Java Bridge Data Type Mapping" on page 147 for more information.

Note

Besides other Java based objects, no complex types (Structures, pointers, etc.) are supported as parameters to property calls.

Destroying IDL-Java Objects

The OBJ_DESTROY routine is used to destroy instances of IDLjavaObject. When OBJ_DESTROY is called with a Java based object as an argument, IDL releases the underlying Java object and frees IDL resources relating to that object.

Note

Destruction of the IDL object does not automatically cause the destruction of the underlying Java object. Because Java utilizes a garbage collection mechanism to release any information allocated for a particular object, the resources utilized by the underlying Java object will persist until the Java virtual machine’s garbage collector runs.

Showing IDL-Java Output in IDL

By default, IDL prints the output from Java (the `System.out` and `System.err` output streams).

For example, given the following Java code:

```
public class helloWorld
{
 // ctor
 public helloWorld() {
  System.out.println("helloWorld ctor");
   }

 public void sayHello() {
  System.out.println("Hello! (from the helloWorld object)");
   }

}
```

The following output occurs in IDL:

```
IDL> oJHello = OBJ_NEW('IDLjavaObject$HelloWorld', 'helloWorld')
% helloWorld ctor
IDL> oJHello -> SayHello
% Hello! (from the helloWorld object)
IDL> OBJ_DESTROY, oJHello
```

This example code is also provided in the `helloJava.java` and `hellojava2.pro` files, which are in the `external/objbridge/java/examples` directory of the IDL distribution.

Note

Due to restrictions in IDL concerning receiving standard output from non-main threads, the bridge will only send `System.out` and `System.err` information to IDL from the main thread. Other thread’s output will be ignored.

Note

A `print()` in Java will not have a carriage return at the end of the line (as opposed to `println()`, which does). However, when outputting to Java both `print()` and `println()` will print to IDL followed by a carriage return. You can change this result by having the Java-side application buffer its data up into the lines you wish to see on the IDL-side.

The IDLJavaBridgeSession Object

Java exceptions are handled within IDL through an IDL-Java bridge session object, IDLJavaBridgeSession. This Java object can be queried to determine the status of the bridge, including information on any exceptions. For example, one important Java object available through the session object is the last issued Java exception.

The session object is a proxy to an internal Java object, which is created during the IDL-Java bridge initialization process. You can connect an IDLJavaObject to this object using OBJ_NEW:

```
oJSession = OBJ_NEW('IDLjavaObject$IDLJAVABRIDGESESSION’)
```

Note

Only one Java session object needs to be created during an IDL session. Subsequent calls to this object will point to the same internal object.

When an exception occurs, the GetException function method indicates what exception occurred:

```
oJException = oJSession -> GetException()
```

where `oJSession` is a reference to the session object and `oJException` is a proxy object to a `java.lang.Throwable` object, which is the class used in Java to manage exceptions. The session object also has a ClearException method that clears the session object’s last exception. The GetException method always calls ClearException method.

The IDLJavaBridgeSession object also has the GetVersionObject method, which retrieves the IDLJavaVersion object:

```
oJVersion = oJSession -> GetVersionObject()
```

where `oJSession` is a reference to the session object and `oJVersion` is a proxy object to an IDLJavaVersion object. This object determines version information about the IDL-Java bridge and the underlying Java system.

The IDLJavaVersion object provides the following function methods, which do not require any arguments:

- GetBuildDate() - a java.lang.String object specifying the build date. For example, `Apr 1 2003`.
- GetJavaVersion() - a java.lang.String object specifying the Java version. For example, `1.3.1_02`.
- GetBridgeVersion() - a java.lang.String object specifying the IDL-Java bridge version.

An example of the version object is provided in the `bridge_version.pro` file, which is in IDL's `external/objbridge/java/examples` directory.

Java Exceptions

During the operation of the bridge, an error may occur when initializing the bridge, creating an `IDLjavaObject`, calling methods, setting properties, or getting properties. Typically, these errors will be fixed by changing your IDL or Java code (or by changing the bridge configuration). Java bridge errors operate like other IDL errors in that they stop execution of IDL and post an error message. These errors can be caught like any other IDL error.

On the other hand, Java uses the exception mechanism to report errors. For example, in Java, if we attempt to create a java.lang.StringBuffer of negative length, a java.lang.NegativeArraySizeException is issued.

Java exceptions are handled much like bridge errors. They stop IDL execution (if uncaught) and they report an error message containing a line number. In addition, a mechanism is provided to grab the exception object (a subclass of java.lang.Throwable) via the session object. Once connected with the exception object, IDL can call any of the methods provided by this Java object. For example, IDL can query the exception name to determine how to handle it, or print a stack trace of where the exception occurred in your Java code.

The exception object is provided through the GetExpection method to the IDLJavaBridgeSession object. See "The IDLJavaBridgeSession Object" on page 161 for more information about this object.

Uncaught Exceptions

If a Java exception is not caught, IDL will stop execution and display an `Exception thrown` error message. For example, when the following program is saved as `ExceptIssued.pro`, compiled, and ran in IDL:

```
PRO ExceptIssued

; This will throw a Java exception
oJStrBuffer = OBJ_NEW($
   'IDLJavaObject$java_lang_StringBuffer', $
   ’java.lang.StringBuffer’, -2)

END
```

IDL issues the following output:

```
IDL> ExceptIssued
% Exception thrown
% Execution halted at: EXCEPTISSUED 4 ExceptIssues.pro
%                      $MAIN$
```

From the IDL command line, you can then use the session object to help debug the problem:

```
IDL> oJSession = OBJ_NEW('IDLJavaObject$IDLJAVABRIDGESESSION')
IDL> oJExc = oJSession -> GetException()
IDL> oJExc -> PrintStackTrace
% java.lang.NegativeArraySizeException:
%    at java.lang.StringBuffer.<init>(StringBuffer.java:116)
```

A similar example is also provided in the `exception.pro` file, which is in the `external/objbridge/java/examples` directory of the IDL distribution. The `exception.pro` example shows how to use the utility routine provided in the `showexcept.pro` file. This `showexcept` utility routine can be re-used to provide consist error messages when Java exceptions occur. The `showexcept.pro` file is also provided in the `external/objbridge/java/examples` directory of the IDL distribution.

Caught Exceptions

Java exceptions can be caught just like IDL errors. Consult the documentation of the Java classes that you are using to ensure IDL is catching any expected exceptions. For example:

```
PRO ExceptCaught

; Grab the special IDLJavaBridgeSession object
oJBridgeSession = OBJ_NEW('IDLJavaObject$IDLJAVABRIDGESESSION')

bufferSize = -2
; Our Java constructor might throw an exception, so let’s catch it
CATCH, error_status
IF (error_status NE 0) THEN BEGIN
   ; Use session object to get our Exception
   oJExc = oJBridgeSession -> GetException()
   ; should be of type
   ; IDLJAVAOBJECT$JAVA_LANG_NEGATIVEARRAYSIZEEXCEPTION
   HELP, oJExc
   ; Now we can access the members java.lang.Throwable
   PRINT, 'Exception thrown:', oJExc -> ToString()
   oJExc -> PrintStackTrace
   ; Cleanup
   OBJ_DESTROY, oJExc
```

```
      ; Increase the buffer size to avoid the exception.
      bufferSize = bufferSize + 100
   ENDIF

   ; This throws a Java exception the 1st time, but pass the 2nd time.
   oJStrBuffer = OBJ_NEW('IDLJavaObject$java_lang_StringBuffer', $
      'java.lang.StringBuffer', bufferSize)


   OBJ_DESTROY, oJStrBuffer
   OBJ_DESTROY, oJBridgeSession

   END
```

A similar example is also provided in the `exception.pro` file, which is in the `external/objbridge/java/examples` directory of the IDL distribution. The `exception.pro` example shows how to use the utility routine provided in the `showexcept.pro` file. This `showexcept` utility routine can be re-used to provide consist error messages when Java exceptions occur. The `showexcept.pro` file is also provided in the `external/objbridge/java/examples` directory of the IDL distribution.

IDL-Java Bridge Examples

The following examples demonstrate how to access data through the IDL-Java bridge:

- "Accessing Arrays Example"
- "Accessing URLs Example" on page 169
- "Accessing Grayscale Images Example" on page 171
- "Accessing RGB Images Example" on page 174

Note

If IDL is not able to find any Java class associated with these examples, make sure your IDL-Java bridge is properly configured. See "Configuring the Bridge" on page 144 for more information.

Accessing Arrays Example

This example creates a two-dimensional array within a Java class, which is contained in a file named `array2d.java`. IDL then accesses this data through the ArrayDemo routine, which is in a file named `arraydemo.pro`. These files are also in the IDL distribution within the `external/objbridge/java/examples` directory.

The `array2d.java` file contains the following text for creating a two-dimensional array in Java:

```
public class array2d
{
 short[][]   m_as;
 long[][]    m_aj;

 // ctor
 public array2d() {
   int SIZE1 = 3;
   int SIZE2 = 4;

   // default ctor creates a fixed number of elements
   m_as = new short[SIZE1][SIZE2];
   m_aj = new long[SIZE1][SIZE2];

   for (int i=0; i<SIZE1; i++) {
     for (int j=0; j<SIZE2; j++) {
       m_as[i][j] = (short)(i*10+j);
       m_aj[i][j] = (long)(i*10+j);
```

```
         }
       }

     }


     public void setShorts(short[][] _as) {
       m_as = _as;
     }
     public short[][] getShorts() {return m_as;}
     public short getShortByIndex(int i, int j) {return m_as[i][j];}


     public void setLongs(long[][] _aj) {
       m_aj = _aj;
     }
     public long[][] getLongs() {return m_aj;}
     public long getLongByIndex(int i, int j) {return m_aj[i][j];}


    }
```

The `arraydemo.pro` file contains the following text for accessing the two-dimensional array within IDL:

```
PRO ArrayDemo

; The Java class array2d creates 2 initial arrays, one
; of longs and one of shorts. We can interrogate and
; change this array.
oJArr = OBJ_NEW('IDLJavaObject$ARRAY2D', 'array2d')

; First, let’s see what is in the short array at index
; (2,3).
PRINT, 'array2d short(2, 3) = ', $
   oJArr -> GetShortByIndex(2, 3), $
   '    (should be 23)’

; Now, let’s copy the entire array from Java to IDL.
shortArrIDL = oJArr -> GetShorts()
HELP, shortArrIDL
PRINT, 'shortArrIDL[2, 3] = ', shortArrIDL[2, 3], $
   '    (should be 23)'

; Let’s change this value...
shortArrIDL[2, 3] = 999
; ...and copy it back to Java...
oJArr -> SetShorts, shortArrIDL
; ...now its value should be different.
```

```
PRINT, 'array2d short(2, 3) = ', $
   oJArr -> GetShortByIndex(2, 3), '    (should be 999)'

; Let’s set our array to something different.
oJArr -> SetShorts, INDGEN(10, 8)
PRINT, 'array2d short(0, 0) = ', $
   oJArr -> GetShortByIndex(0, 0), '    (should be 0)'
PRINT, 'array2d short(1, 0) = ', $
   oJArr -> GetShortByIndex(1, 0), '    (should be 1)'
PRINT, 'array2d short(2, 0) = ', $
   oJArr -> GetShortByIndex(2, 0), '    (should be 2)'
PRINT, 'array2d short(0, 1) = ', $
   oJArr -> GetShortByIndex(0, 1), '    (should be 10)'

; Array2d has a setLongs method, but b/c arrays do not
; (currently) promote, the first call to setLongs works
; but the second fails.
oJArr -> SetLongs, L64INDGEN(10, 8)
PRINT, 'array2d long(0, 1) = ', $
   oJArr -> GetLongByIndex(0, 1), '    (should be 10)'

;PRINT, '(expecting an error on the next line...)'
;oJArr -> SetLongs, INDGEN(10,8)

; Cleanup our object.
OBJ_DESTROY, oJArr

END
```

After saving and compiling the above files (`array2d.java` in Java and `ArrayDemo.pro` in IDL), update the `jbexamples.jar` file in the `external/objbridge/java` directory with the new compiled class and run the ArrayDemo routine in IDL. The routine should produce the following results:

```
array2d short(2, 3) = 23 (should be 23)
SHORTARRIDL   INT = Array[3, 4]
shortArrIDL[2, 3] = 23 (should be 23)
array2d short(2, 3) = 999 (should be 999)
array2d short(0, 0) = 0 (should be 0)
array2d short(1, 0) = 1 (should be 1)
array2d short(2, 0) = 2 (should be 2)
array2d short(0, 1) = 10 (should be 10)
array2d long(0, 1) = 10 (should be 10)
```

Accessing URLs Example

This example finds and reads a given URL, which is contained in a file named `URLReader.java`. IDL then accesses this data through the URLRead routine, which is in a file named `urlread.pro`. These files are also in the IDL distribution within the `external/objbridge/java/examples` directory.

The `URLReader.java` file contains the following text for reading a given URL in Java:

```
import java.io.*;
import java.net.*;

public class URLReader
{
  private ByteArrayOutputStream m_buffer;

  // ***********************************************************
  //
  // Constructor.  Create the reader
  //
  // ***********************************************************
   public URLReader() {
      m_buffer = new ByteArrayOutputStream();
   }

  // ***********************************************************
  //
  // readURL: read the data from the URL into our buffer
  //
  //    returns: number of bytes read (0 if invalid URL)
  //
  // NOTE: reading a new URL clears out the previous data
  //
  // ***********************************************************
   public int readURL(String sURL) {
      URL url;
      InputStream in = null;


      m_buffer.reset();  // reset our holding buffer to 0 bytes

      int total_bytes = 0;
      byte[] tempBuffer = new byte[4096];
      try {
         url = new URL(sURL);
         in = url.openStream();
```

```
            int bytes_read;
            while ((bytes_read = in.read(tempBuffer)) != -1) {
               m_buffer.write(tempBuffer, 0, bytes_read);
               total_bytes += bytes_read;
            }
         } catch (Exception e) {
            System.err.println("Error reading URL: "+sURL);
            total_bytes = 0;
         } finally {
            try {
               in.close();
               m_buffer.close();
            } catch (Exception e) {}
         }

         return total_bytes;
      }

     // ********************************************************
     //
     // getData: return the array of bytes
     //
     // ********************************************************
      public byte[] getData() {
        return m_buffer.toByteArray();
      }

     // ********************************************************
     //
     // main: reads URL and reports # of byts reads
     //
     //   Usage: java URLReader <URL>
     //
     // ********************************************************

     public static void main(String[] args) {
        if (args.length != 1)
           System.err.println("Usage: URLReader <URL>");
        else {
           URLReader o = new URLReader();
           int b = o.readURL(args[0]);
           System.out.println("bytes="+b);
        }
     }



   }
```

The `urlread.pro` file contains the following text for inputting an URL as an IDL string and then accessing its data within IDL:

```
FUNCTION URLRead, sURLName

; Create an URLReader.
oJURLReader = OBJ_NEW('IDLjavaObject$URLReader', 'URLReader')

; Read the URL data into our Java-side buffer.
nBytes = oJURLReader -> ReadURL(sURLName)

;PRINT, 'Read ', nBytes, ' bytes'

; Pull the data into IDL.
byteArr = oJURLReader -> GetData()

; Cleanup Java object.
OBJ_DESTROY, oJURLReader

; Return the data.
RETURN, byteArr

END
```

After saving and compiling the above files (`URLReader.java` in Java and `urlread.pro` in IDL), you can run the URLRead routine in IDL. This routine is a function with one input argument, which should be a IDL string containing an URL. For example:

```
address = 'http://www.RSInc.com'
data = URLRead(address)
```

Accessing Grayscale Images Example

This example creates a a grayscale ramp image within a Java class, which is contained in a file named `GreyBandsImage.java`. IDL then accesses this data through the ShowGreyImage routine, which is in the `showgreyimage.pro` file. These files are also in the IDL distribution within the `external/objbridge/java/examples` directory.

The `GreyBandsImage.java` file contains the following text for creating a grayscale image in Java:

```
import java.awt.*;
import java.awt.image.*;

public class GreyBandsImage extends BufferedImage
{
 // Members
 private int m_height;
 private int m_width;


 //
 // ctor
 //
 public GreyBandsImage() {
    super(100, 100, BufferedImage.TYPE_INT_ARGB);
    generateImage();
    m_height = 100;
    m_width = 100;
 }

 //
 // private method to generate the image
 //
 private void generateImage() {
    Color c;
    int width  = getWidth();
    int height = getHeight();
    WritableRaster raster = getRaster();
    ColorModel model = getColorModel();

    int BAND_PIXEL_WIDTH = 5;
    int nBands = width/BAND_PIXEL_WIDTH;
    int greyDelta = 255 / nBands;
    for (int i=0 ; i < nBands; i++) {
          c = new Color(i*greyDelta, i*greyDelta, i*greyDelta);
          int argb = c.getRGB();
          Object colorData = model.getDataElements(argb, null);

          for (int j=0; j < height; j++)
             for (int k=0; k < BAND_PIXEL_WIDTH; k++)
                raster.setDataElements(j, (i*5)+k, colorData);

    }
 }
```

```
  //
  // mutators
  //
  public int[] getRawData() {
    Raster oRaster = getRaster();
    Rectangle oBounds = oRaster.getBounds();
    int[] data = new int[m_height * m_width * 4];

    data = oRaster.getPixels(0,0,100,100, data);
    return data;
  }
  public int getH() {return m_height; }
  public int getW() {return m_width; }


}
```

The `showgreyimage.pro` file contains the following text for accessing the grayscale image within IDL:

```
PRO ShowGreyImage

; Construct the GreyBandImage in Java. This is a sub-class of
; BufferedImage. It is actually a 4 band image that happens to
display bands in greyscale. It is 100x100 pixels.
oGrey = OBJ_NEW('IDLjavaObject$GreyBandsImage', 'GreyBandsImage')

; Get the 4 byte pixel values.
data = oGrey -> GetRawData()

; Get the height and width.
h = oGrey -> GetH()
w = oGrey -> GetW()

; Display the graphic in an IDL window
WINDOW, 0, XSIZE = 100, YSIZE = 100
TV, REBIN(data, h, w)

; Cleanup
OBJ_DESTROY, oGrey

END
```

After saving and compiling the above files (`GreyBandsImage.java` in Java and `showgreyimage.pro` in IDL), you can run the ShowGreyImage routine in IDL. The routine should produce the following image:

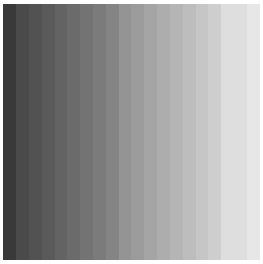


Figure 8-1: Java Grayscale Image Example

Accessing RGB Images Example

This example imports an RGB (red, green, and blue) image from the IDL distribution into a Java class. The image is in the `glowing_gas.jpg` file, which is in the `examples/data` directory of the IDL distribution. The Java class also displays the image in a Java Swing user-interface. Then, the image is accessed into IDL and displayed with the new iImage tool. The Java and IDL code for this example is provided in the `external/objbridge/java/examples` directory, but the Java code has not been built as part of the `jbexamples.jar` file.

Note

This example uses functionality only available in Java 1.4 and later.

Note

Due to a Java bug, this example (and any other example using Swing on AWT) will not work on Linux platforms.

The first and main Java class is FrameTest, which creates the Java Swing application that imports the image from the `glowing_gas.jpg` file. Copy and paste the following text into a file, then save it as `FrameTest.java`:

```
import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.io.File;

public class FrameTest extends JFrame {

 RSIImageArea c_imgArea;
 int m_xsize;
 int m_ysize;
 Box c_controlBox;

 public FrameTest() {

  super("This is a JAVA Swing Program called from IDL");
  // Dispose the frame when the sys close is hit
  setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  m_xsize = 350;
  m_ysize = 371;
  buildGUI();

 }

 public void buildGUI() {

  c_controlBox = Box.createVerticalBox();

  JLabel l1 = new JLabel("Example Java/IDL Interaction");
  JButton bLoadFile = new JButton("Load new file");
  bLoadFile.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    JFileChooser chooser = new JFileChooser(new
     File("c:\\RSI\\IDL60\\EXAMPLES\\DATA"));
    chooser.setDialogTitle("Enter a JPEG file");
    if (chooser.showOpenDialog(FrameTest.this) ==
     JFileChooser.APPROVE_OPTION) {

      java.io.File fname = chooser.getSelectedFile();
      String filename = fname.getPath();
      System.out.println(filename);
      c_imgArea.setImageFile(filename);
    }
   }
  });

  JButton b1 = new JButton("Close this example");
  b1.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    dispose();
   }
  });

  c_imgArea = new
   RSIImageArea("c:\\rsi\\idl60\\examples\\data\\glowing_gas.jpg",
    new Dimension(m_xsize,m_ysize));



  Box mainBox = Box.createVerticalBox();
  Box rowBox = Box.createHorizontalBox();
  rowBox.add(b1);
  rowBox.add(bLoadFile);

  c_controlBox.add(l1);
  c_controlBox.add(rowBox);
  mainBox.add(c_controlBox);
  mainBox.add(c_imgArea);

  getContentPane().add(mainBox);

  pack();
  setVisible(true);
  c_imgArea.displayImage();
  c_imgArea.addResizeListener(new RSIImageAreaResizeListener() {
   public void areaResized(int newx, int newy) {
    Dimension cdim = c_controlBox.getSize(null);
    Insets i = getInsets();
    newx = i.left + i.right + newx;
    newy = i.top + cdim.height + newy + i.bottom;
    setSize(new Dimension(newx, newy));
   }
  });
 }

 public void setImageData(int [] imgData, int xsize, int ysize) {
  MemoryImageSource ims = new MemoryImageSource(xsize, ysize,
   imgData, 0, ysize);
  Image imgtmp = createImage(ims);
  Graphics g = c_imgArea.getGraphics();
  g.drawImage(imgtmp, 0, 0, null);

 }
```

```
public void setImageData(byte [][][] imgData, int xsize,
 int ysize) {


 System.out.println("SIZE = "+xsize+"x"+ysize);
 int newArray [] = new int[xsize*ysize];
 int pixi = 0;
 int curpix = 0;
 short [] currgb = new short[3];
 for (int i=0;i<m_xsize;i++) {
  for (int j=0;j<m_ysize;j++) {
   for (int k=0;k<3;k++) {
    currgb[k] = (short) imgData[k][i][j];
    currgb[k] = (currgb[k] < 128) ? (short) currgb[k] : (short)
     (currgb[k]-256);
   }
   curpix = (int) currgb[0] *  +
    ((int) currgb[1] * (int) Math.pow(2,8)) +
     ((int) currgb[2] * (int) Math.pow(2,16));
   if (pixi % 1000 == 0)
    System.out.println("PIXI = "+pixi+" "+curpix);
   newArray[pixi++] = curpix;
  }
 }

 MemoryImageSource ims = new MemoryImageSource(xsize, ysize,
  newArray, 0, ysize);
 c_imgArea.setImageObj(c_imgArea.createImage(ims));

}

public byte[][][] getImageData()
{
 int width = 1;
 int height = 1;
 PixelGrabber pGrab;

 width = m_xsize;
 height = m_ysize;

 // pixarray for the grab - 3D bytearray for display
 int [] pixarray = new int[width*height];
 byte [][][] bytearray = new byte[3][width][height];


 // create a pixel grabber
 pGrab = new PixelGrabber(c_imgArea.getImageObj(),0,0,
      width,height, pixarray, 0, width);

      // grab the pixels from the image
      try {
       boolean b = pGrab.grabPixels();
      } catch (InterruptedException e) {
       System.err.println("pixel grab interrupted");
       return bytearray;
      }

      // break down the 32-bit integers from the grab into 8-bit bytes
      // and fill the return 3D array
      int pixi = 0;
      int curpix = 0;
      for (int j=0;j<m_ysize;j++) {
       for (int i=0;i<m_xsize;i++) {
        curpix = pixarray[pixi++];
        bytearray[0][i][j] = (byte) ((curpix >> 16) & 0xff);
        bytearray[1][i][j] = (byte) ((curpix >>  8) & 0xff);
        bytearray[2][i][j]  = (byte) ((curpix      ) & 0xff);
       }
      }
      return bytearray;
     }


     public static void main(String [] args) {
      FrameTest f = new FrameTest();
     }

    }
```

Note

The above text is for the FrameTest class that accesses the `glowing_gas.jpg` file in the `examples/data` directory of a default installation of IDL on a Windows system. The file's location is specified as `c:\\RSI\\IDL60\\EXAMPLES\\DATA` in the above text. If the `glowing_gas.jpg` file is not in the same location on system, edit the text to change the location of this file to match your system.

The FrameTest class uses two other user-defined classes, RSIImageArea and RSIImageAreaResizeListener. These classes help to define the viewing area and display the image in Java. Copy and paste the following text into a file, then save it as `RSIImageArea.java`:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;
import java.io.File;

public class RSIImageArea extends JComponent implements
 MouseMotionListener, MouseListener {


 Image c_img;
 int m_boxw = 100;
 int m_boxh = 100;
 Dimension c_dim;
 boolean m_pressed = false;
 int m_button = 0;
 Vector c_resizelisteners = null;

 public RSIImageArea(String imgFile, Dimension dim) {

  c_img = getToolkit().getImage(imgFile);
  c_dim = dim;
  setPreferredSize(dim);
  setSize(dim);
  addMouseMotionListener(this);
  addMouseListener(this);

 }

 public void addResizeListener(RSIImageAreaResizeListener l) {
  if (c_resizelisteners == null) c_resizelisteners = new Vector();
  if (! c_resizelisteners.contains(l))  c_resizelisteners.add(l);
 }
 public void removeResizeListener(RSIImageAreaResizeListener l) {
  if (c_resizelisteners == null) return;
  if (c_resizelisteners.contains(l)) c_resizelisteners.remove(l);
 }

 public void displayImage() {
  repaint();
 }

 public void paint(Graphics g) {
  int xsize = c_img.getWidth(null);
  int ysize = c_img.getHeight(null);
  if (xsize != -1 && ysize != -1) {
   if (xsize != c_dim.width || ysize != c_dim.height) {
    c_dim.width = xsize;
    c_dim.height = ysize;
    setPreferredSize(c_dim);
    setSize(c_dim);
    if (c_resizelisteners != null) {
     RSIImageAreaResizeListener l = null;
     for (int j=0;j<c_resizelisteners.size();j++) {
      l = (RSIImageAreaResizeListener)
       c_resizelisteners.elementAt(j);
      l.areaResized(xsize, ysize);
     }
    }
   }
  }
  g.drawImage(c_img, 0, 0, null);
 }

 public void setImageFile(String fileName) {
  c_img = null;
  c_img = getToolkit().getImage(fileName);
  repaint();
 }


 public Image getImageObj() {
  return c_img;
 }

 public void setImageObj(Image img) {
  c_img = img;
  repaint();
 }

 public void drawZoomBox(MouseEvent e) {
  int bx = e.getX() - m_boxw/2;
  bx = (bx >=0) ? bx :0;
  int by = e.getY() - m_boxh/2;
  by = (by >=0) ? by :0;
  int ex = bx + m_boxw;
  if (ex > c_dim.width) {
   ex = c_dim.width;
   bx = c_dim.width-m_boxw;
  }
  int ey = by + m_boxh;
  if (ey > c_dim.height) {
```

```
  ey = c_dim.height;
  by = c_dim.height-m_boxh;
 }

 repaint();
 Graphics g = getGraphics();
 g.drawImage(c_img, bx, by, ex, ey, bx+(m_boxw/4), by+(m_boxh/4),
  ex-(m_boxw/4),ey-(m_boxh/4), null);
 g.setColor(Color.white);
 g.drawRect(bx, by, m_boxw, m_boxh);

}

public void mouseDragged(MouseEvent e) {
 drawZoomBox(e);
}

public void mouseMoved(MouseEvent e) {

 Graphics g = getGraphics();
 if (m_pressed && (m_button == 1)) {
  drawZoomBox(e);
  g.setColor(Color.white);
  g.drawString("DRAG", 10,10);
 } else {

  g.setColor(Color.white);
  String s = "("+e.getX()+","+e.getY()+")";
  repaint();
  g.drawString(s, e.getX(), e.getY());
 }

}

public void mouseClicked(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}

public void mousePressed(MouseEvent e) {
 m_pressed = true;
 m_button = e.getButton();
 repaint();
 if (m_button == 1) drawZoomBox(e);
}

public void mouseReleased(MouseEvent e) {
 m_pressed = false;
 m_button = 0;
}
```

```
}
```

And copy and paste the following text into a file, then save it as `RSIImageAreaResizeListener.java`:

```
public interface RSIImageAreaResizeListener {
 public void areaResized(int newx, int newy);
}
```

Compile these classes in Java. Then, either update the `jbexamples.jar` file in the `external/objbridge/java` directory with the new compiled class, place the resulting compiled classes in your Java class path, or edit the JVM Classpath setting in the IDL-Java bridge configuration file to specify the location (path) of these compiled classes. See “Configuring the Bridge” on page 144 for more information.

With the Java classes compiled, you can now access them in IDL. Copy and paste the following text into the IDL Editor window, then save it as `ImageFromJava.pro`:

```
PRO ImageFromJava
; Create a Swing Java object and have it load image data
; into IDL.

; Create the Java object first.
oJSwing = OBJ_NEW('IDLjavaObject$FrameTest', 'FrameTest')

; Get the image from the Java object.
image = oJSwing -> GetImageData()
PRINT, 'Loaded Image Information:'
HELP, image

; Delete the Java object.
OBJ_DESTROY, oJSwing

; Interactively display the image.
IIMAGE, image

END
```

After compiling the above routine, you can run it in IDL. This routine produces the following Java Swing application.

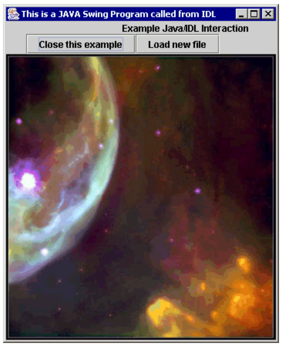
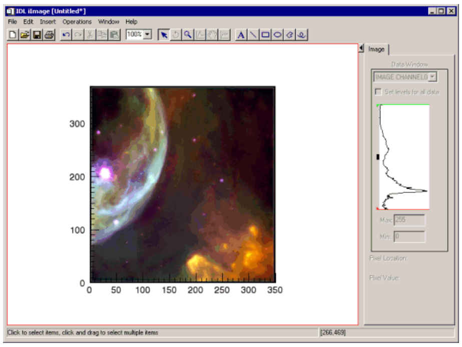

This is a JAVA Swing Program called from IDL
Example Java/IDL Interaction
Close this example
Load new file

Figure 8-2: Java Swing Application Example

Then, the routine produces the following iImage tool.



Click to select items, click and drag to select multiple items [266,469]

Figure 8-3: iImage Tool from Java Swing Example

Note

After IDL starts the Java Swing application, the two displays are independent of each other. If a new image is loaded into the Java application, the IDL iImage tool is not updated. If the iImage tool modifies the existing image or opens a new image, the Java Swing application is not updated.

Troubleshooting Your Bridge Session

The IDL-Java bridge provides error messages for specific types of operations. These messages can be used to determine when these errors occur, how these errors happen, and what solutions can be applied. The following sections pertain to these error messages and their possible solutions for each type of operation:

- "Errors when Initializing the Bridge"
- "Errors when Creating Objects" on page 185
- "Errors when Calling Methods" on page 186
- "Errors when Accessing Data Members" on page 187

Errors when Initializing the Bridge

The IDL-Java bridge initializes when the first Java object in IDL is created. If the bridge is not configured correctly, an error message is issued and the IDL stops. The following errors occur because the IDL-Java bridge cannot find the Java Virtual Machine on your system. On UNIX, check the $IDLJAVAB_LIB_LOCATION environment variable, and on Windows, check the IDLJAVAB_LIB_LOCATION environment variable. If this environment variable does not exist on your system, create it and set it equal to the location of the Java Virtual Machine on your system. See "Configuring the Bridge" on page 144 for details:

- `Bad JVM Home value: '`*path*`'`, where *path* is the location of Java Virtual Machine on your system.
- *JVM shared lib* `not found in path '`*JVM LibLocation*`'`, where *JVM shared lib* is the location of the Java Virtual Machine shared library and *JVM LibLocation* is the value of the IDLJAVAB_LIB_LOCATION environment variable.
- `No valid JVM shared library exists at location pointed to by $IDLJAVAB_LIB_LOCATION`
- `idljavab.jar not found in path '`*path*`'`, where *path* is the location of the `external/objbridge/java` directory in the IDL distribution.
- `Bridge cannot determine which JVM to run`

- `Java virtual machine failed to start`
- `Failure loading JVM:` *`path/JVM shared lib name`*, where *path* is the location of the Java Virtual Machine and *JVM shared lib name* is the name of the main Java shared library, which is usually `libjvm.so` on UNIX and `jvm.dll` on Windows.

If IDL catches an error and continues, subsequent attempts to call the bridge will generate the following message:

- `IDL-Java bridge is not running`

If this message occurs, fix the error and restart IDL.

Errors when Creating Objects

The following error messages can occur while creating a Java object in IDL. Possible solutions for these errors are also provided:

- `Wrong number of parameters` - occurs if OBJ_NEW does not have 2 or more parameters. Make sure you are specifying the class name twice; once in uppercase with periods replaced by underscores for IDL, and another with periods for Java. See "Java Class Names in IDL" on page 153 for details.
- `Second parameter must be the Java class name` - occurs if 2nd parameter is not an IDL string. When using OBJ_NEW, make sure the Java class name parameter is an IDL string. In other words, the class name has a single quote mark before and after it. See "Java Class Names in IDL" on page 153 for details.
- `Class` *classname* `not found`, where *classname* is the class name you specified in the first two parameters to OBJ_NEW - occurs if the IDL-Java bridge cannot find the class name specified. Check the spelling of each class name parameter and make sure the class name specified for IDL is referring to the same type of object specified for the Java class name. If the parameters are correct, check the Classpath setting in the IDL-Java bridge configuration file. Make sure the Classpath is set to the correct path for the class files containing the *classname* class. See "Configuring the Bridge" on page 144 for details.

- `Class` *classname* `is not a public class`, where *classname* is the class name you specified in the first two parameters to OBJ_NEW - occurs if specified class is not a public class. Edit your Java code to make sure the class you want to access is public.
- `Constructor` *class*`::`*class*`(`*signature*`) not found`, where *class* is the class name - occurs if the IDL-Java bridge cannot find the class constructor with the given parameters. Check the spelling of the specified parameters and look in your Java code to see if you are specifying the correct arguments for the class you are trying to create. Also check to ensure your IDL data can be promoted to the data types in the Java signature. See "Java Class Names in IDL" on page 153 for details.
- `Illegal IDL value in parameter` *n*, where *n* is the position of the parameter - occurs if an illegal parameter type is provided. For example, an IDL structure is not allowed as a parameter to an IDLjavaObject.
- `Exception thrown` - occurs if an exception occurs in Java. Either correct or handle the Java exception. The Java exception can be determined with the IDLJavaBridgeSession object. See "The IDLJavaBridgeSession Object" on page 161 for details.

Errors when Calling Methods

The following error messages can occur while calling methods to Java objects in IDL. Possible solutions for these errors are also provided:

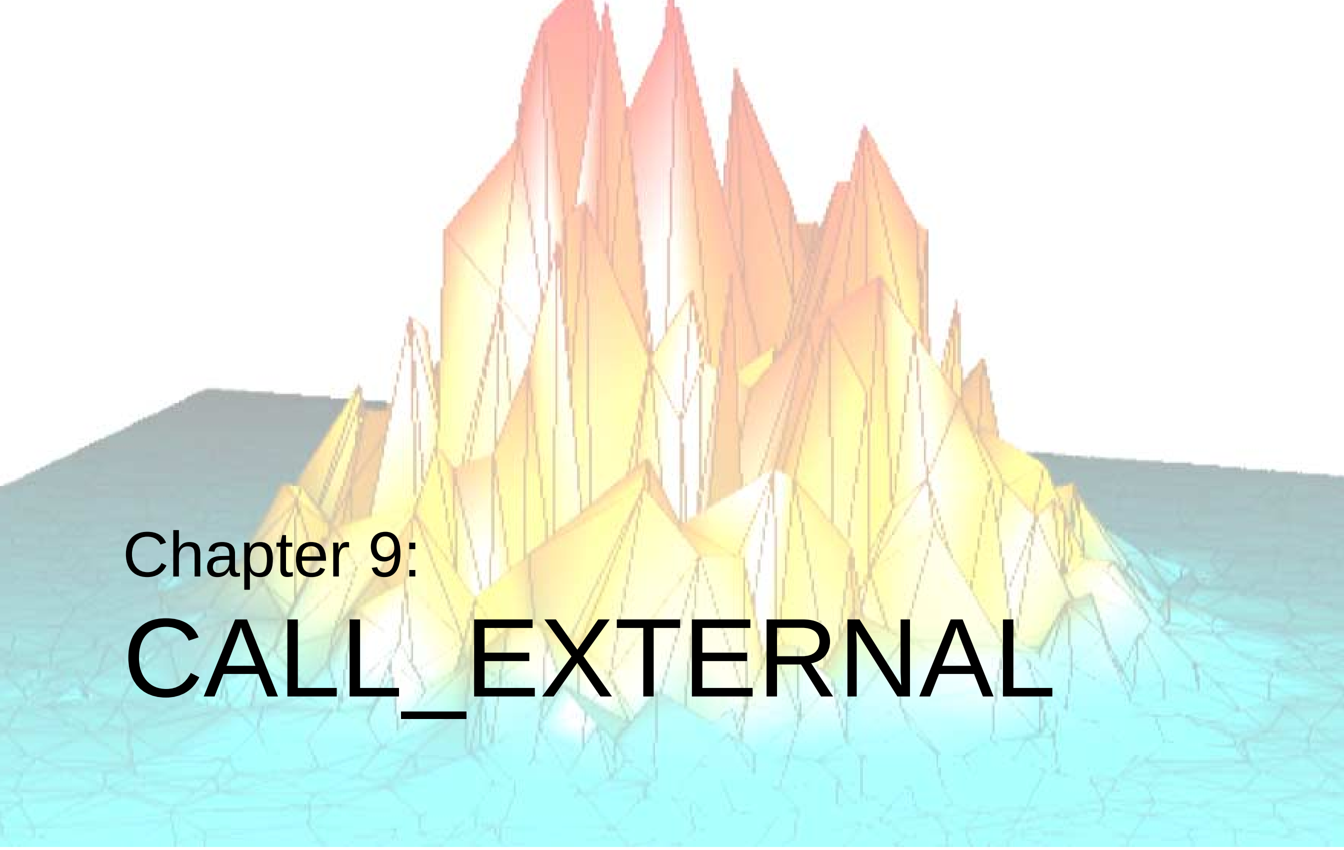
- `Illegal IDL value in parameter` *n*, where *n* is the position of the parameter - occurs if an illegal parameter type is provided. For example, an IDL structure are not allowed as a parameter to an IDLjavaObject.
- `Class` *class* `has no method named` *method*, where *class* is the class name and *method* is the method name specified when trying to call the Java method - occurs if the method of given name does not exist. Check the spelling of the method name. Also compare the method name in the Java class source file with the method name provided when calling the method in IDL. See "What Happens When a Method Call is Made?" on page 155 for details.
- *class*`::`*method*`(`*signature*`) is a void method. Must be called as a procedure`, where *class* is the class name and *method* is the method name specified when a void Java method is called as an IDL function. Change the syntax of the method call. See "Method Calls on IDL-Java Objects" on page 155 for details.

- `Method` *class*`::`*method*`(`*signature*`) not found`, where *class* is the class name and *method* is the method name specified when trying to call the Java method - occurs if the IDL-Java bridge cannot find the method with a matching signature. Check the spelling of the method name. Also compare the method name in the Java class source file with the method name provided when calling the method in IDL. Also check to ensure your IDL data can be promoted to the Java signature. See "What Happens When a Method Call is Made?" on page 155 for details.
- `Exception thrown` - occurs if an exception occurs in Java. Either correct or handle the Java exception. The Java exception can be determined with the IDLJavaBridgeSession object. See "The IDLJavaBridgeSession Object" on page 161 for details.

Errors when Accessing Data Members

The following error messages can occur while accessing data members to Java objects in IDL. Possible solutions for these errors are also provided:

- `Illegal IDL value in parameter` *n*, where *n* is the position of the parameter - occurs if an illegal parameter type is provided. For example, an IDL structure is not allowed as a parameter to an IDLjavaObject.
- `Class` *class* `has no data member named` *property*, where *class* is the class name and *property* is the data member name specified when trying to access the Java data member - occurs if the data member of the given name does not exist. Check the spelling of the property name. Also compare the data member name in the Java class source file with the property name provided when accessing it in IDL. See "Managing IDL-Java Object Properties" on page 157 for details.
- `Property` *class*`::`*property* `of type` *type* `not found`, where *class* is the class name, *property* is the data member name specified, and *type* is *property*'s data type when trying to access the Java data member - occurs if the IDL-Java bridge cannot find the Java data member of the given type. Check the data type of Java data member and make sure you are trying to use a similar type in IDL. See "Getting and Setting Properties" on page 158 for details.
- `Exception thrown` - occurs if an exception occurs in Java. Either correct or handle the Java exception. The Java exception can be determined with the IDLJavaBridgeSession object. See "The IDLJavaBridgeSession Object" on page 161 for details.



Chapter 9: CALL_EXTERNAL

This chapter discusses the following topics:

The CALL_EXTERNAL Function

IDL allows you to integrate programs written in other languages with your IDL code, either by calling a compiled function from an IDL program or by linking a compiled function into IDL's internal system routine table:

- The CALL_EXTERNAL function allows you to call external functions (written in C/C++ or Fortran, for example) from your IDL programs. You should be comfortable writing and building programs in the external language being used, but significant knowledge of IDL's internals beyond basic type mapping between the languages is generally not necessary.
- An alternative to CALL_EXTERNAL is to write an IDL system routine and merge it with IDL at runtime. Routines merged in this fashion are added to IDL's internal system routine table and are available in the same manner as IDL built-in routines. This technique is discussed in Chapter 21, "Adding System Routines". To write a system routine, you will need to understand the IDL internals discussed in later sections of this book.

This chapter covers the basics of using CALL_EXTERNAL from IDL, then discusses platform-specific options for the UNIX and Windows versions of IDL. It can be helpful to refer to the documentation for "CALL_EXTERNAL" in the *IDL Reference Guide* manual when reading this material.

The CALL_EXTERNAL function loads and calls routines contained in shareable object libraries. Arguments passed to IDL are passed to this external code, and returned data from the external code is automatically presented as the result from CALL_EXTERNAL as an IDL variable. IDL and the called routine share the same process address space. Because of this, CALL_EXTERNAL avoids the overhead of process creation of the SPAWN routine. In addition, the shareable object library is only loaded the first time it is referenced, saving overhead on subsequent calls.

CALL_EXTERNAL is much easier to use than writing a system routine. Unlike a system routine, however, CALL_EXTERNAL does not check the type or number of parameters. Programming errors in the external routine are likely to result in corrupted data (either in the routine or in IDL) or to cause IDL to crash. See "Common CALL_EXTERNAL Pitfalls" on page 197 for help in avoiding some of the more common mistakes.

Example Code in the IDL Distribution

This chapter contains examples of CALL_EXTERNAL use. All of the code for these examples, along with additional examples, can be found in the `call_external` subdirectory of the `external` directory of the IDL distribution. The C language examples use the MAKE_DLL procedure, and can therefore be easily run on any platform supported by IDL. To build the sharable library containing the external C code and then run all of the provided examples, execute the following IDL statements:

```
PUSHD, FILEPATH('',SUBDIRECTORY=['external','call_external','C'])
ALL_CALLEXT_EXAMPLES
POPD
```

Additional information on these examples, including details on running the individual examples, can be found in the README file located in that directory.

CALL_EXTERNAL Compared To UNIX Child Process

In many situations, a UNIX IDL user has a choice of using the SPAWN procedure to start a child process that executes external code and communicates with IDL via a pipe connecting the two processes. The advantages of this approach are:

- Simplicity.
- The processes do not share address space, and are therefore protected from each other's mistakes.

The advantages of CALL_EXTERNAL are:

- IDL and the called routine share the same memory and data space. Although this can be a disadvantage (as noted above) there are times where sharing address space is advantageous. For example, large data can be easily and cheaply shared in this manner.
- CALL_EXTERNAL avoids the overhead of process creation and parameter passing.
- The shareable object library containing the called routine is only loaded the first time it is referenced, whereas a SPAWNed process must be created for each use of the external code.

Compilation and Linking Of External Code

Each operating system requires different compilation and link statements for producing a shareable object suitable for usage with CALL_EXTERNAL. This is even true between different implementations of a common operating system family. For example, most UNIX systems require unique options despite their shared heritage. You must consult your system and compiler documentation to find the appropriate options for your system.

The IDL MAKE_DLL procedure, documented in the *IDL Reference Guide*, provides a portable high level mechanism for building sharable libraries from code written in the C programming language. In many situations, this procedure can completely handle the task of building sharable libraries to be used with CALL_EXTERNAL. MAKE_DLL requires that you have a C compiler installed on your system that is compatible with the compiler described by the IDL !MAKE_DLL system variable.

The IDL !MAKE_DLL system variable is used by the MAKE_DLL procedure to construct C compiler and linker commands appropriate for the target platform. If you do not use MAKE_DLL to compile and link your code, you may find the contents of !MAKE_DLL.CC and !MAKE_DLL.LD helpful in determining which options to specify to your compiler and linker, in conjunction with your system and compiler documentation. For the C language, the options in !MAKE_DLL should be very close to what you need. For other languages, the !MAKE_DLL options should be helpful in determining which options to use, as on most systems, all the language compilers accept similar options.

AUTO_GLUE

As described in "Passing Parameters" on page 200, CALL_EXTERNAL uses the *IDL Portable Calling Convention* to call external code. This convention uses an (`argc`, `argv`) style interface to allow CALL_EXTERNAL to call routines with arbitrary numbers and types of arguments. Such an interface is necessary, because IDL, like any compiled program, cannot generate arbitrary function calls at runtime.

Of course, most C functions are not written to the IDL portable convention. Rather, they are written using the natural form of argument passing used in compiled programs. It is therefore common for IDL programmers to write so-called *glue functions* to match the IDL calling interface to that of the target function. On systems that have a C compiler installed that is compatible with the one described by the IDL !MAKE_DLL system variable, the AUTO_GLUE keyword to CALL_EXTERNAL can be used to instruct IDL to automatically write, compile, and load this glue code on demand, and using a cache to preserve this glue code for future invocations of functions with the same interface.

AUTO_GLUE thus allows CALL_EXTERNAL to call functions with a natural interface, without requiring the user to write or compile additional code. AUTO_GLUE is described in the documentation for "CALL_EXTERNAL" in the *IDL Reference Guide* manual, as well as in "Using Auto Glue" on page 202. The examples given in "Basic C Examples" on page 204 show CALL_EXTERNAL used with and without AUTO_GLUE.

Input and Output

Input and output actions should be performed within IDL code, using IDL's built-in input/output facilities, or by using **IDL_Message()**. Performing input/output from code external to IDL, especially to the user console or tty (e.g. `stdin` or `stdout`), may generate unexpected results.

Memory Cleanup

IDL has a strict internal policy that it never performs memory cleanup on memory that it did not allocate. This policy is necessary so that external code which allocates memory can use any memory allocation package it desires, and so that there is no confusion about which code is responsible for releasing allocated memory.

Note

The code that allocates memory is always responsible for freeing it. IDL allocates and frees memory for its internal needs, and external code is not allowed to release such memory except through a proper IDL function documented for that purpose. Similarly, IDL will never intentionally free memory that it did not allocate.

As such, IDL does not perform any memory cleanup calls on the values returned from external code called via the CALL_EXTERNAL routine. Because of this, any dynamic memory returned to IDL will not be returned to the system, which will result in a memory leak. Users should be aware of this behavior and design their CALL_EXTERNAL routines in such a manner as not to return dynamically allocated memory to IDL. The discussion in "Passing String Data" on page 209 contains an example of doing this with strings.

Memory Access

IDL and your external code share the same address space within the same running program. This means that mistakes common in compiled languages, such as a wild pointer altering memory that it does not own, can cause problems elsewhere. In particular, external code can easily corrupt IDL's data structures and otherwise cause IDL to fail. Authors of such code must be especially careful to guard against such errors.

Argument Data Types

When using CALL_EXTERNAL to call external code, IDL passes its arguments to the called code using the data types that were passed to it. It has no way to verify independently that these types are the actual types expected by the external routine. If the data types passed are not of the types expected by the external code, the results are undefined, and can easily include memory corruption or even crashing of the IDL program.

Warning

You must ensure that the arguments passed to external code are of the exact type expected by that routine. Failure to do so will result in undefined behavior.

Mapping IDL Data Types To External Language Types

When writing external code for use with CALL_EXTERNAL, your code must use data types that are compatible with the C data types used internally by IDL to represent the IDL data types. This mapping is the topic of Chapter 11, “IDL Internals: Types”.

By-Value And By-Reference Arguments

There are two basic forms in which arguments can be passed between functions in compiled languages such as C/C++ and Fortran. To use CALL_EXTERNAL successfully, you should be comfortable with these terms and their meanings. In particular, Fortran programmers are often unaware that Fortran code passes everything by reference, and that C code defaults to passing everything by value. By default, CALL_EXTERNAL passes arguments by reference (unless this behavior is explicitly altered by the use of the ALL_VALUE or VALUE keywords), so no special action is typically required to call Fortran code via CALL_EXTERNAL.

Warning

You must ensure that the arguments passed to external code are passed using the correct method — by value, or by reference. Failure to do so will result in undefined behavior.

Arguments Passed By Value

A copy of the value of the argument is passed to the called routine. Any changes made to such a value by the called routine are local to that routine, and do not change the original value of the variable in the calling routine. C/C++ pass everything by value, but have an explicit *address-of* operator (&) that is used to pass addresses of variables and get by-reference behavior.

Arguments Passed By Reference

The machine address of the argument is passed to the called routine. Any changes made to such a value by the called routine are immediately visible to the caller, because both routines are actually modifying the same memory addresses. Fortran passes everything by reference, but most Fortran implementations support intrinsic operators that allow the programmer control over this (sometimes called %LOC and %VAL, or just LOC and VAL). Consult your compiler documentation for details.

Microsoft Windows Calling Conventions

All operating system/hardware combinations define an inter-routine calling convention. A *calling convention* defines the rules used for passing arguments between routines, and specifies such details as how arguments of different types are passed (*i.e.* in registers or on the system stack) and how and when such arguments are cleaned up.

A stable and efficient calling convention is critical to the stability of an operating system, and can affect most aspects of the system:

- The efficiency of the entire system depends on the efficiency of the core calling convention.
- Backwards compatibility, and thus the longevity of binary software written for the platform depends on the stability of the calling convention.
- Calling routines from different languages within a single program depends on all the language compilers adhering to the same calling convention. Even within the same language, the ability to mix code compiled by different compilers requires those compilers to adhere to the same conventions. For example, at the time of this writing, the C++ language standard lacks an Application Binary Interface (ABI) that can be targeted by all C++ compilers. This can lead to situations in which the same compiler must be used to build all of the code within a given program.

Microsoft Windows is unique among the platforms supported by IDL in that it has two distinct calling conventions in common use, whereas other systems define a single convention. On single-convention systems, the calling convention is unimportant to application programmers, and of concern only to hardware designers and the authors of compilers, and operating systems. On a multiple convention system, application programmers sometimes need to be aware of the issue, and ensure that their code is compiled to use the proper convention and that calls to that code use the same convention. The Microsoft Calling Conventions are:

STDCALL

STDCALL is the calling convention used by the majority of the Windows operating system API. In a STDCALL call, the calling routine places the arguments in the proper registers and/or stack locations, and the called routine is responsible for cleaning them up and unwinding the stack.

CDECL

CDECL is the calling convention used by C/C++ code by default. This default can be changed via compiler switches, declspec declarations, or #pragmas. With CDECL, the caller is responsible for both setup and cleanup of the arguments. CDECL is able to call functions with variable numbers of arguments (*varargs* functions) because the caller knows the actual number of arguments passed at runtime, whereas STDCALL cannot call such functions. This is because the STDARGS routine cannot know efficiently at compile time how many arguments it will be passed at runtime in these situations.

The inconvenience of having two distinct and incompatible calling conventions is usually minor, because the header files that define functions for C/C++ programs include the necessary definitions such that the compiler knows to generate the proper code to call them and the programmer is not required to be aware of the issue. However, CALL_EXTERNAL does have a problem: Unlike a C/C++ program, IDL determines how to call a function solely by the arguments passed to CALL_EXTERNAL, and not from a header file. IDL therefore has no way to know how your external code was compiled. It uses the STDARG convention by default, and the CDECL keyword can be used to change the default. CALL_EXTERNAL therefore relies on the IDL user to tell it which convention to use. If IDL calls your code using the correct convention, it will work correctly. If it calls using the wrong convention, the results are undefined, including memory corruption and possible crashing of the IDL program.

Warning

The default calling convention for CALL_EXTERNAL is STDCALL, whereas the default convention for the Microsoft C compiler is CDECL. Hence, Windows users must usually specify the CDECL keyword when calling such code from IDL. Non-Windows versions of IDL ignore the CDECL keyword, so it is safe to always include it in cross platform code.

Here is what happens when external code is called via the wrong calling convention:

- If a STDARG call is made to a CDECL function, the caller places the arguments in the proper registers/stack locations, and relies on the called routine to cleanup and unwind the stack. The called routine, however, does not do these things because it is a CDECL routine. Hence, cleanup does not happen.

- If a CDECL call is made to a STDARG function, the caller places the arguments in the proper register/stack locations. The called routine cleans up on exit, and then the caller cleans up again.

Either combination is bad, and can corrupt or kill the program. Sometimes this happens, and sometimes it doesn't, so the results can be random and mysterious to programmers who are not aware of the issue.

Note

When the wrong calling convention is used, it is common for the process stack to become confused. A "smashed stack" visible from the C debugger following a CALL_EXTERNAL is usually indicative of having used the wrong calling convention.

Common CALL_EXTERNAL Pitfalls

Following are a list of common errors and mistakes commonly seen when using CALL_EXTERNAL.

- The number of arguments and their types, as passed to CALL_EXTERNAL, must be the exact types expected by the external routine. In particular, it is common for programmers to forget that the default IDL integer is a 16-bit value and that most C compilers define the `int` type as being a 32-bit value. You should be careful to use IDL LONG integers, which are 32-bit, in such cases. See "Argument Data Types" on page 193 for additional details.
- Passing data using the wrong form: Using by-value to pass an argument to a function expecting it by-reference, or the reverse. See"By-Value And By-Reference Arguments" on page 194 for additional details.
- Under Microsoft Windows, using the incorrect calling convention for a given external function. See "Microsoft Windows Calling Conventions" on page 195 for additional details.
- Failure to understand that IDL uses IDL_STRING descriptors to represent strings, and not just a C style NULL terminated string. Passing a string value by reference passes the address of the IDL_STRING descriptor to the external code. See Chapter 14, "IDL Internals: String Processing" for additional details.
- Attempting to make IDL data structures use memory allocated by external code rather than using the proper IDL API for creating such data structures.

For instance, attempting to give an **IDL_STRING** descriptor a different value by using C **malloc()** to allocate memory for the string and then storing the address of that memory in the **IDL_STRING** descriptor is not supported, and can easily crash or corrupt IDL. Although IDL uses **malloc()/free()** internally on most platforms, you should be aware that this is not part of IDL's public interface, and that RSI can change this at any time and without notice. Even on platforms where IDL does use these functions, its use of them is not directly compatible with similar calls made by external code because IDL allocates additional memory for bookkeeping that is generally not present in memory allocations from other sources. See Chapter 14, "IDL Internals: String Processing" for information on changing the value of an IDL_STRING descriptor using supported IDL interfaces. See Chapter 9, "Memory Cleanup" for more on memory allocation and cleanup.

- IDL is written in the C language, and when IDL starts, any necessary runtime initialization code required by C programs is automatically executed by the system before the IDL **main()** function is called. Hence, calling C code from IDL usually does not require additional runtime initialization. However, when calling external code written in languages other than C, you may find that your code does not run properly unless you arrange for the necessary runtime support for that language to run first. Such details are highly system specific, and you must refer to your system and compiler documentation for details. Code that is largely computational rarely encounters this issue. It is more common for code that performs Input/Output directly.
- Programming errors in the external code. It is easy to make mistakes in compiled languages that have bad global consequences for unrelated code within the same program. For example, a wild memory pointer can lead to the corruption of unrelated data. If you are lucky, such an error will immediately kill your program, making it easy to locate and fix. Less fortunate is the situation in which the program dies much later in a seemingly unrelated part of the program. Finding such problems can be difficult and time consuming. When IDL crashes following a call to external code, an error in the external code or in the call to CALL_EXTERNAL is the cause in the vast majority of cases.
- Some compilers and operating systems have a convention of adding leading or trailing underscore characters to the names of functions they compile. These conventions are platform specific, and as they are of interest only to system linker and compiler authors, not generally well documented. This is usually transparent to the user, but can sometimes be an issue with inter language function calls. If you find that a function you expect to call from a library is not being found by CALL_EXTERNAL, and the obvious checks do not uncover

the error (usually a simple misspelling), this might be the cause. Under UNIX, the **nm** command can be helpful in diagnosing such problems.

- C++ compilers use a technique commonly called *name munging* to encode the types of method arguments and return values into the name of the routine as written to their binary object files. Such names often have only a passing resemblance to the name seen by the C++ programmer in their source code. IDL can only call C++ code that has C linkage, as discussed in "C++" on page 25. C linkage code does not use name munging.
- When calling external code written in other languages, there are sometimes platform and language specific hidden arguments that must be explicitly supplied. Such arguments are usually provided by the compiler when you work strictly within the target language, but become visible in inter-language calls. An example of this can be found in "Hidden Arguments" on page 218. In this example, the Fortran compiler provides an extra hidden length argument when a NULL terminated string is passed to a function.

Passing Parameters

IDL calls routines within in a shareable library using the *IDL portable calling convention*, in which the routine is passed two arguments:

argc

A count of the number of arguments being passed to the routine

argv

An array of **argc** memory pointers, which are the addresses of the arguments (by reference) or the actual value of the argument (by value) depending on the types of arguments passed to CALL_EXTERNAL and the setting of the VALUE keyword to that function. You should note that while all types of data can be passed by reference, there are limitations on data types that can be passed by value, as described in the documentation for "CALL_EXTERNAL" in the *IDL Reference Guide* manual.

The CALL_EXTERNAL portable convention is necessary because IDL, like any program written in a compiled language, cannot generate arbitrary function calls at runtime. Only calls to interfaces that were known to it when it was compiled are possible. Naturally, most existing C functions are not written to use this interface. Calling such functions typically requires IDL users to write *glue functions*, the sole purpose of which is to be called by CALL_EXTERNAL with the portable convention, and then to take the arguments and pass them to the real target function using the natural interface for that function. The AUTO_GLUE keyword to CALL_EXTERNAL can be used to generate, compile, and load such glue routines automatically and on demand, without requiring user intervention. Auto Glue is described in "Using Auto Glue" on page 202. AUTO_GLUE does not eliminate the need for, or use of, the portable convention, but it can relieve the IDL user of the requirement to handle it explicitly. The end result is that calling existing function interfaces is easier to do, and less error prone.

Routines called by CALL_EXTERNAL with the portable convention are defined with a prototype similar to the following:

```
return_type example(int argc; void *argv[])
```

where *return_type* is one of the data types which CALL_EXTERNAL can return. If this *return_type* is not IDL_LONG, a keyword must be used in the CALL_EXTERNAL call to indicate the actual type of the result.

The parameter `argc` gives the number of arguments passed to the external routine by CALL_EXTERNAL in the `argv` array, while `argv` is an array containing the arguments. Arguments are passed either by value or by reference. Those passed by value are copied directly into the `argv` array, with the exception of scalar strings, which place a pointer to a null-terminated string in `argv[i]`. All arrays are passed by reference. Scalar items passed by reference (the default) place a pointer to the datum in `argv[i]`. Strings and string arrays passed by reference place a pointer to an IDL_STRING structure in `argv[i]`. This structure is defined as follows:

```
typedef struct {
  IDL_STRING_SLEN_T slen;   /* Length of string */
  short stype;  /* type of string:  (0) static, (!0) dynamic */
  char *s;      /* Addr of string, invalid if slen == 0.  */
} IDL_STRING;
```

See "CALL_EXTERNAL" in the *IDL Reference Guide* manual for additional details about passing parameters by value.

It is important to note that IDL integer variables correspond to a 16-bit integer (a C *`signed`* short integer). For example, an integer variable could be defined in an IDL routine as follows:

```
IDL> A = 5          ;default type of integer, not LONG
```

The variable could then be passed by reference in a CALL_EXTERNAL call. The declaration and cast statement in the called C routine should be:

```
short *a;
a = (short *) argv[0];
```

or

```
IDL_INT *a;
a = (IDL_INT *) argv[0];
```

IDL_INT corresponds to a C short (16-bit integer), so either form is correct. The corresponding type in Fortran would be INTEGER*2.

Using Auto Glue

Users of CALL_EXTERNAL frequently write small functions with the sole purpose of matching the CALL_EXTERNAL portable calling convention with its (`argc`, `argv`) interface to the actual interface presented by some existing function that they wish to call. Such functions are often called *glue functions*.

It quickly becomes obvious to anyone who has written a few glue functions that there isn't much to them, and that producing such functions is a purely mechanical operation. As you read the examples in this chapter, you will see many such functions, and will notice that they are all essentially the same. Further examination should serve to convince you that IDL already has all of the information, in the form of the arguments and keywords specified to the CALL_EXTERNAL function, to generate such functions without requiring human intervention. Examining the CALL_EXTERNAL routine's interface, we see that:

- the number and types of arguments to the CALL_EXTERNAL function provide the same information about the arguments for the target external function;
- the VALUE keyword, and CALL_EXTERNAL's built in rules for deciding whether or not to pass arguments by value or by reference determine how the arguments should be passed;
- in the case of Microsoft Windows, the CDECL keyword tells it which system calling convention to employ;
- keywords to CALL_EXTERNAL determine the result type.

Furthermore, other than the actual name of the user function being called, these glue functions are generic in the sense that they could be used to call any function that accepted arguments of the same types and produce a result of the same type.

The AUTO_GLUE keyword to CALL_EXTERNAL exploits these facts to allow you to call functions with natural interfaces, without the need to write, compile, and load a glue function to do the job. The sole requirement is that your system must have a C compiler installed that is compatible with the compiler described by the IDL !MAKE_DLL system variable. This is almost always the case if you are interested in calling external code, since a compiler is necessary to compile such code.

AUTO_GLUE automatically writes the C code for the glue function, uses the MAKE_DLL procedure to build a sharable library containing it, loads that library, and then calls the glue function, passing it a pointer to the target function and all of its arguments. It maintains a cache of glue functions that have been built previously, and never builds the same glue function more than once. From the user perspective, there is a slight pause the first time a given glue function is used. In that brief moment, AUTO_GLUE performs the steps described above, and then makes the call to the user function. All of this happens transparently to the IDL user — no user interaction is required, and no output is produced by the process. Subsequent calls to the same glue function happen instantaneously, as IDL loads the existing glue function from the MAKE_DLL cache without rebuilding it. In principle, it is similar to the way IDL automatically compiles IDL language programs on demand, only with C code instead of IDL code.

See “CALL_EXTERNAL” in the *IDL Reference Guide* manual for additional details about how AUTO_GLUE works, and the options for controlling its use.

Generating Glue Without Executing It

AUTO_GLUE is the preferred option for most calls to functions with natural interfaces, due to it’s simplicity and ease of use. However, you might find yourself in a situation where you would like your glue functions to be automatically generated, but wish to simply get the resulting C code so that you can modify it or incorporate it into a larger library. For example, you might have a large library of IDL specific code, and wish to give it all IDL callable interfaces without requiring the overhead of AUTO_GLUE for all of them.

The WRITE_WRAPPER keyword to CALL_EXTERNAL can be used to produce such code without compiling or using the results. See “CALL_EXTERNAL” in the *IDL Reference Guide* manual for additional information on this keyword.

Basic C Examples

All of the code for the examples in this section can be found in the `/external/call_external/C` subdirectory of the IDL distribution. Please read the README file in that directory for details on how to run the examples. In many cases, the files in that directory go into more detail, and are more fully commented than the versions shown here. Also, the examples provide IDL wrapper routines that perform the necessary CALL_EXTERNAL calls, while the examples shown here use CALL_EXTERNAL directly in order to explain how it is used. It is worth reading the contents of the `.c` and IDL `.pro` files in that directory in addition to reading the code shown here.

Example: Passing Parameters by Reference to IDL

The following routine, found in `simple_vars.c`, accepts several of IDL's basic data types as arguments. The parameters are passed in by reference and the new squared values of the numbers are passed back to IDL. This is implemented as a function with a natural C interface, and a second glue routine that implements the IDL portable convention, using the one with the natural interface to do the actual work.

The IDL statements necessary to call the `simple_vars()` function from IDL can be written:

```
B=2B & I=3 & L=3L & F=0.0 & D=0.0D
R = CALL_EXTERNAL(GET_CALLEXT_EXLIB(), 'simple_vars', $
                  b,i,l,f,d, /CDECL)
```

Note

`GET_CALLEXT_EXLIB()` is a function provided with the CALL_EXTERNAL examples; it builds the necessary sharable library of external C code and returns the path to the library as its result.

Using the AUTO_GLUE keyword to CALL_EXTERNAL, you can call the function with the natural C interface directly:

```
B=2B & I=3 & L=3L & F=0.0 & D=0.0D
R = CALL_EXTERNAL(GET_CALLEXT_EXLIB(), 'simple_vars_natural', $
                  b,i,l,f,d, /CDECL, /AUTO_GLUE)
```

Example: Calling a C Routine to Perform

C

```c
#include <stdio.h>
#include "idl_export.h"         /* IDL external definitions */

int simple_vars_natural(char *byte_var, short *short_var,
                        IDL_LONG *long_var, float *float_var,
                        double *double_var)
{
  /* Square each variable. */
  *byte_var     *= *byte_var;
  *short_var    *= *short_var;
  *long_var     *= *long_var;
  *float_var    *= *float_var;
  *double_var   *= *double_var;

  return 1;
}

int simple_vars(int argc, void* argv[])
{
  /* Insure that the correct number of arguments were passed in */
  if(argc != 5) return 0;

  return simple_vars_natural((char *) argv[0], (short *) argv[1],
                             (IDL_LONG *) argv[2], (float *) argv[3],
                             (double *) argv[4]);
}
```

Table 9-1: Passing Parameters by Reference to IDL — simple_vars.c

Computation

The following example demonstrates an external function that returns the sum of a floating point array. It is similar in function to the TOTAL function in IDL. The code for this example is found in the file `sum_array.c` in the IDL distribution. As with the previous example, this function is implemented by a function that has a natural C interface, and a second glue function is provided that matches the IDL portable calling convention to the natural interface:

C

```
#include <stdio.h>
#include "idl_export.h"

float sum_array_natural(float *fp, IDL_LONG n)
{
  float s = 0.0;

  while (n--) s += *fp++;
  return(s);
}

float sum_array(int argc, void *argv[])
{
  return sum_array_natural((float *) argv[0], (IDL_LONG) argv[1]);
}
```

Table 9-2: Calling a C routine — example.c

The IDL statements necessary to call the `sum_array()` function from IDL can be written:

```
X = FINDGEN(10)
S = CALL_EXTERNAL(GET_CALLEXT_EXLIB(), 'sum_array'$
                  X, N_ELEMENTS(X),VALUE=[0,1], /F_VALUE, /CDECL)
```

Note

`GET_CALLEXT_EXLIB()` is a function provided with the CALL_EXTERNAL examples; it builds the necessary sharable library of external C code and returns the path to the library as its result.

Using the AUTO_GLUE keyword, you can call the function with the natural C interface directly:

```
X = FINDGEN(10)
S = CALL_EXTERNAL(GET_CALLEXT_EXLIB(), 'sum_array_natural'$
                  X, N_ELEMENTS(X),VALUE=[0,1], /F_VALUE,/CDECL,$
                  /AUTO_GLUE)
```

In this example, `sum_array` and `sum_array_natural` are the names of the entry points for the external functions, and `X` and `N_ELEMENTS(X)` are passed to the called routine as parameters. The `F_VALUE` keyword specifies that the returned value is a floating-point number rather than an IDL_LONG.

Wrapper Routines

CALL_EXTERNAL routines are very sensitive to the number and type of the arguments they receive. Calling a CALL_EXTERNAL routine with the wrong number of arguments or with arguments of the wrong type can cause IDL to crash. For this reason, it is a good practice to provide an IDL *wrapper routine* that is used to make the actual CALL_EXTERNAL call. The job of this wrapper, which is written in the IDL language, is to ensure that the arguments that are passed to the external code are always of the correct number and type. The following IDL procedure is the wrapper used in the **simple_vars()** example of the previous section ("Example: Passing Parameters by Reference to IDL" on page 204). It can be found in the IDL distribution in the file `simple_vars.pro`.

IDL

```
PRO SIMPLE_VARS, b, i, l, f, d, AUTO_GLUE=auto_glue, DEBUG=debug, $
        VERBOSE=verbose
  if ~ (KEYWORD_SET(debug)) THEN ON_ERROR,2

  ; Type checking: Any missing (undefined) arguments will be set
  ; to a default value. All arguments will be forced to a scalar
  ; of the appropriate type, which may cause errors to be thrown
  ; if structures are passed in. Local variables are used so that
  ; the values and types of the user supplied arguments don’t change.
  b_l = (SIZE(b,/TYPE) EQ 0) ? 2b   : byte(b[0])
  i_l = (SIZE(i,/TYPE) EQ 0) ? 3    : fix(i[0])
  l_l = (SIZE(l,/TYPE) EQ 0) ? 4L   : long(l[0])
  f_l = (SIZE(f,/TYPE) EQ 0) ? 5.0  : float(f[0])
  d_l = (SIZE(d,/TYPE) EQ 0) ? 6.0D : double(d[0])

  PRINT, ’Calling simple_vars with the following arguments:’
  HELP, b_l, i_l, l_l, f_l, d_l
  func = keyword_set(auto_glue) ? ’simple_vars_natural’ : ’simple_vars’
  IF (CALL_EXTERNAL(GET_CALLEXT_EXLIB(VERBOSE=verbose), func, $
                    b_l, i_l, l_l, f_l, d_l, /CDECL, $
                    AUTO_GLUE=auto_glue, VERBOSE=verbose, $
                    SHOW_ALL_OUTPUT=verbose) EQ 1) then BEGIN
    PRINT,’After calling simple_vars:’
    HELP, b_l, i_l, l_l, f_l, d_l
  ENDIF ELSE MESSAGE,’External call to simple_vars failed’
END
```

Table 9-3: Wrapper Routine — simple_vars.pro

The routine `simple_vars.pro` uses the system routine SIZE() to examine the arguments that are passed in by the user to the `simple_vars` routine. If one of the arguments is undefined, a default value will be used in the call to the external routine. Otherwise, the argument will be converted to a scalar of the appropriate type.

Note

`GET_CALLEXT_EXLIB()` is a function provided with the CALL_EXTERNAL examples; it builds the necessary sharable library of external C code and returns the path to the library as its result.

Passing String Data

IDL represents strings internally as IDL_STRING descriptors. For more information about IDL_STRING, see Chapter 12, "IDL Internals: Variables" and Chapter 14, "IDL Internals: String Processing". These descriptors are defined in the C language as:

```
typedef struct {
  IDL_STRING_SLEN_T slen;
  unsigned short stype;
  char *s;
} IDL_STRING;
```

To pass a string by reference, IDL passes the address of its IDL_STRING descriptor. To pass a string by value the string pointer (the `s` field of the descriptor) is passed. Programmers should be aware of the following when manipulating IDL strings:

- Called code should treat the information in the passed IDL_STRING descriptor and the string itself as read-only, and should not modify these values.
- The `slen` field contains the length of the string without including the NULL termination that is required at the end of all C strings.
- The `stype` field is used internally by IDL to keep track of how the memory for the string was obtained, and should be ignored by CALL_EXTERNAL users.
- `s` is the pointer to the actual C string represented by the descriptor. If the string is NULL, IDL represents it as a NULL (0) pointer, not as a pointer to an empty null terminated string. Hence, called code that expects a string pointer should check for a NULL pointer before dereferencing it.
- You must use the functions discussed in Chapter 14, "IDL Internals: String Processing" to allocate the memory for an IDL_STRING. Attempting to do this directly by allocating dynamic memory and assigning it to the IDL_STRING descriptor is a common pitfall, as discussed in "Common CALL_EXTERNAL Pitfalls" on page 197.

Returning a String Value

When returning a string value, a function must allocate the memory used to hold it. On return, IDL will copy this string. You can use a static buffer or dynamic memory, but do not return the address of an automatic (stack-based) variable.

Note

IDL will not free dynamically-allocated memory for this use.

Example

The following routine, found in `string_array.c`, demonstrates how to handle string variables in external code. This routine takes a string or array of strings as input and returns a copy of the longest string that it received. It is important to note that this routine uses a static `char` array as its return value, which avoids the possibility of a memory leak, but which must be long enough to handle the longest string required by the application. This is implemented as a function with a natural C interface, and a second glue routine that implements the IDL portable convention, using the one with the natural interface to do the actual work:

C

```
#include <stdio.h>
#include <string.h>
#include "idl_export.h"
/*
 * IDL_STRING is declared in idl_export.h like this:
 *   typedef struct {
 *   IDL_STRING_SLEN_T slen;        Length of string, 0 for null
 *   short stype;                   Type of string, static or dynamic
 *   char *s;                       Address of string
 * } IDL_STRING;
 * However, you should rely on the definition in idl_export.h instead
 * of declaring your own string structure.
*/

char* string_array_natural(IDL_STRING *str_descr, IDL_LONG n)
{
  /*
  * IDL will make a copy of the string that is returned (if it is
  * not NULL). One way to avoid a memory leak is therefore to return
  * a pointer to a static buffer containing a null terminated string. IDL
  * will copy the contents of the buffer and drop the reference to our
  * buffer immediately on return.
  */
#define MAX_OUT_LEN 511         /* truncate any string longer than this */
  static char result[MAX_OUT_LEN+1];    /* leave a space for a '\0' on the
                                           longest string */
  int max_index;                /* index of longest string */
  int max_sofar;                /* length of longest string*/
  int i;

  /*  Check the size of the array passed in. n should be > 0.*/
  if (n < 1) return (char *) 0;
  max_index = 0;
  max_sofar = 0;
  for(i=0; i < n; i++) {
    if (str_descr[i].slen > max_sofar) {
      max_index = i;
      max_sofar = str_descr[i].slen;
    }
  }
```

Figure 9-1: Handling String Variables in External Code — string_array.c

C

```
 /*
  * If all strings in the array are empty, the longest
  * will still be a NULL string.
  */
  if (str_descr[max_index].s == NULL) return (char *) 0;

  /*
  * Copy the longest string into the buffer, up to MAX_OUT_LEN characters.
  * Explicitly store a NULL byte in the last byte of the buffer, because
  * strncpy() does not NULL terminate if the string copied is truncated.
  */
  strncpy(result, str_descr[max_index].s, MAX_OUT_LEN);
  result[sizeof(result)-1] = '\0';
  return(result);
#undef MAX_OUT_LEN
}

char* string_array(int argc, void* argv[])
{
  /*
  * Make sure there are the correct  # of arguments.
  * IDL will convert the NULL into an empty string ('').
  */
  if (argc != 2) return (char *) NULL;
  return string_array_natural((IDL_STRING *) argv[0], (IDL_LONG) argv[1]);
}
```

Figure 9-1: (Continued) Handling String Variables in External Code — string_array.c

Passing Array Data

When you pass an IDL array into a CALL_EXTERNAL routine, that routine gets a pointer to the first memory location in the array. In order to perform any processing on the array, an external routine needs more information—such as the array's size and number of dimensions. With CALL_EXTERNAL, you will need to pass this information explicitly as additional arguments to the routine.

In order to handle multi-dimensional arrays, C needs to know the size of the array at compile time. In most cases, this means that you will need to treat multi-dimensional arrays passed in from IDL as one dimensional arrays. However, you can still build your own indices to access an array as if it had more than one dimension in C. For example, the IDL array index:

```
array[x,y]
```

could be represented in a CALL_EXTERNAL routine as:

```
array_ptr[x + x_size*y];
```

The following routine, found in `sum_2d_array.c`, calculates the sum of a subsection of a two dimensional array. This is implemented as a function with a natural C interface, and a second glue routine that implements the IDL portable convention, using the one with the natural interface to do the actual work:

C

```c
#include <stdio.h>
#include "idl_export.h"
double sum_2d_array_natural(double *arr, IDL_LONG x_start, IDL_LONG x_end,
                            IDL_LONG x_size, IDL_LONG y_start,
                            IDL_LONG y_end, IDL_LONG y_size)
  /* Since we didn’t know the dimensions of the array at compile time, we
   *must treat the input array as if it were a one dimensional vector. */
  IDL_LONG x,y;
  double result = 0.0;

  /* Make sure that we don’t go outside the array.strictly speaking, this
   *is redundant since identical checks are performed in the IDL wrapper
   * routine.IDL_MIN() and IDL_MAX() are macros from idl_export.h */
  x_start = IDL_MAX(x_start,0);
  y_start = IDL_MAX(y_start,0);
  x_end = IDL_MIN(x_end,x_size-1);
  y_end = IDL_MIN(y_end,y_size-1);

  /* loop through the subsection */
  for (y = y_start;y <= y_end;y++)
    for (x = x_start;x <= x_end;x++)
      result += arr[x + y*x_size]; /* build the 2d index: arr[x,y] */
  return result;
}

double sum_2d_array(int argc,void* argv[])
{
  if (argc != 7) return 0.0;
  return sum_2d_array_natural((double *) argv[0], (IDL_LONG) argv[1],
                              (IDL_LONG) argv[2], (IDL_LONG) argv[3],
                              (IDL_LONG) argv[4], (IDL_LONG) argv[5],
                              (IDL_LONG) argv[6]);
}
```

Table 9-4: Adding the Elements of a 2D IDL Array — sum_2d_array.c

The IDL system routine interface provides much more support for the manipulation of IDL array variables. See Chapter 21, “Adding System Routines” for more information.

Passing Structures

IDL structure variables are stored in memory in the same layout that C uses. This makes it possible to pass IDL structure variables into CALL_EXTERNAL routines, as long as the layout of the IDL structure is known. To access an IDL structure from an external routine, you must create a C structure definition that has the exact same layout as the IDL structure you want to process.

For example, for an IDL structure defined as follows:

```
s = {ASTRUCTURE,zero:0B,one:0L,two:0.0,three:0D,four: intarr(2)}
```

the corresponding C structure would look like the following:

```
typedef struct {
   unsigned char zero;
   IDL_LONG one;
   float two;
   double three;
   short four[2];
} ASTRUCTURE;
```

Then, cast the pointer from *argv* to the structure type, as follows:

```
ASTRUCTURE* mystructure;
mystructure = (ASTRUCTURE*) argv[0];
```

The following routine, found in `incr_struct.c`, increments each field of an IDL structure of type ASTRUCTURE. This is implemented as a function with a natural C interface, and a second glue routine that implements the IDL portable convention, using the one with the natural interface to do the actual work:

C

```c
#include <stdio.h>
#include "idl_export.h"

/*
 * C definition for the structure that this routine accepts.The
 * corresponding IDL structure definition would look like this:
 *     s = {zero:0B,one:0L,two:0.,three:0D,four: intarr(2)}
*/
typedef struct {
  unsigned char zero;
  IDL_LONG one;
  float two;
  double three;
  short four[2];
} ASTRUCTURE;

int incr_struct_natural(ASTRUCTURE *mystructure, IDL_LONG n)
{
  /* for each structure in the array, increment every field */
  for (; n--; mystructure++) {
    mystructure->zero++;
    mystructure->one++;
    mystructure->two++;
    mystructure->three++;
    mystructure->four[0]++;
    mystructure->four[1]++;
  }

  return 1;
}
int incr_struct(int argc, void *argv[])
{
  if (argc != 2) return 0;
  return incr_struct_natural((ASTRUCTURE*) argv[0], (IDL_LONG) argv[1]);
}

```

Table 9-5: Accessing an IDL Structure from a C Routine — incr_struct.c

It is not possible to access structures with arbitrary definitions using the CALL_EXTERNAL interface. The system routine interface, discussed in Chapter 21, “Adding System Routines”, does provide support for determining the layout of a structure at runtime.

Fortran Examples

Example: Calling a Fortran Routine Using a C Interface Routine

Calling Fortran is similar to calling C, with the significant difference that Fortran code expects all arguments to be passed by reference and not by value (the C default). This means that the *address* of the argument is passed rather than the argument itself. This issue is discussed in "By-Value And By-Reference Arguments" on page 194.

A C interface routine can easily extract the addresses of the arguments from the `argv` array and pass them to the actual routine which will compute the sum. The arguments *f*, *n*, and *s* are pointers that are being passed by value. Fortran expects all arguments to be passed by reference — that is, it expects all arguments to be addresses. If C passes a pointer (an address) by value, Fortran will interpret it correctly as the address of an argument. The following code segments illustrate this. The `example_c2f.c` file contains the C interface routine, which would be compiled as illustrated above. The `example.f` file contains the Fortran routine that actually sums the array.

In these examples, we assume that the routines are being compiled under Sun Solaris. The object name of the Fortran subroutine will be `sum_array1_` to match the output of the Solaris Fortran compiler. The following are the contents of `example_c2f.c` and `example.f`:

C

```
#include <stdio.h>

void sum_array(int argc, void *argv[])
{
  extern void sum_array1_();/* Fortran routine */
  int *n;
  float *s, *f;

  f = (float *) argv[0];    /* Array pntr */
  n = (int *) argv[1];      /* Get # of elements */
  s = (float *) argv[2];    /* Pass back result a parameter */

  sum_array1_(f, n, s);     /* Compute sum */
}
```

Table 9-6: C Wrapper Used to Call Fortran Code (example_c2f.c)

f77	
	```
	c This subroutine is called by SUM_ARRAY and has no IDL-specific code.
	c
	SUBROUTINE sumarray1(array, n, sum)
	INTEGER*4 n
	REAL*4 array(n), sum

	sum=0.0
	DO i=1,n
	sum = sum + array(i)
	PRINT *, sum, array(i)
	ENDDO

	RETURN
	END
	```

Table 9-7: Fortran Code Called from IDL via C Wrapper (example.f)

This example is compiled and linked in a manner similar to that used in the C example above. For more information on compiling and linking on your platform, see the README file contained in the `external/call_external/Fortran` subdirectory of the IDL distribution. This directory also contains a makefile, which builds this example on UNIX platforms. To call the example program from within IDL:

```
;Make an array.
X = FINDGEN(10)
;A floating result
SUM = 0.0
S = CALL_EXTERNAL('example.so', $
   'sum_array', X, N_ELEMENTS(X), sum)
```

In this example, `example.so` is the name of the sharable image file, `sum_array` is the name of the entry point, and *`X`* and *`N_ELEMENTS(X)`* are passed to the called routine as parameters. The returned value is contained in the variable `sum.`

Hidden Arguments

When passing C null-terminated character strings into a Fortran routine, the C function should also pass in the string length. This extra parameter is added to the end of the Fortran routine call in the C function, but does not explicitly appear in the Fortran routine.

For example, in C:

```
char * str1= 'IDL';
char * str2= 'RSI';
int len1=3;
int len2=3;
double data, info;
```

```
/* Call a Fortran sub-routine named example1 */
example1_(str1, data, str2, info, len1, len2)
```

In Fortran:

```
SUBROUTINE EXAMPLE1(STR1, DATA, STR2, INFO)
CHARACTER*(*)STR1, STR2
DOUBLE PRECISIONDATA, INFO
```

Example: Calling a Fortran Routine Using a Fortran Interface Routine

Calling Fortran is similar to calling C, with the significant difference that Fortran expects all arguments to be passed by reference. This means that the *address* of the argument is passed rather than the argument itself. See "By-Value And By-Reference Arguments" on page 194 for more on this subject.

A Fortran interface routine can be written to extract the addresses of the arguments from the `argv` array and pass them to the actual routine which will compute the sum. Passing the contents of each `argv` element by value has the same effect as converting the parameter to a normal Fortran parameter.

This method uses the OpenVMS Extensions to Fortran, %LOC and %VAL. On IBM AIX, the LOC function is an intrinsic operator. The syntax of the call, which differs from that used on other platforms, is:

```
y=loc(x)
```

Some Fortran compilers may not support these extensions. If your compiler does not, use the method discussed in the previous section for calling Fortran with a C interface routine.

The contents of the file `example1.f` are shown in the following figure. This example is compiled, linked, and called in a manner similar to that used in the C example above. For more information on compiling and linking on your platform, see the README file contained in the `external/fortran` subdirectory of the IDL distribution. This directory also contains a makefile, which builds this example on UNIX platforms.

Note

This example is written to run under a 32-bit operating system. To run the example under a 64-bit operating system would require modifications; most notably, to declare `argv` as `INTEGER*8` rather than `INTEGER*4`.

f77

```
SUBROUTINE SUM_ARRAY(argc, argv)  !Called by IDL
INTEGER*4 argc, argv(*)           !Argc and Argv are integers

j = LOC(argc)         !Obtains the number of arguments (argc)
                      !Because argc is passed by VALUE.

c Call subroutine SUM_ARRAY1, converting the IDL parameters
c to standard Fortran, passed by reference arguments:

CALL SUM_ARRAY1(%VAL(argv(1)), %VAL(argv(2)), %VAL(argv(3)))
RETURN
END

c This subroutine is called by SUM_ARRAY and has no
c IDL specific code.
c
SUBROUTINE SUM_ARRAY1(array, n, sum)
INTEGER*4 n
REAL*4 array(n), sum

sum=0.0
DO i=1,n
sum = sum + array(i)
ENDDO
RETURN
END
```

Table 9-8: Fortran Code Called Directly From IDL

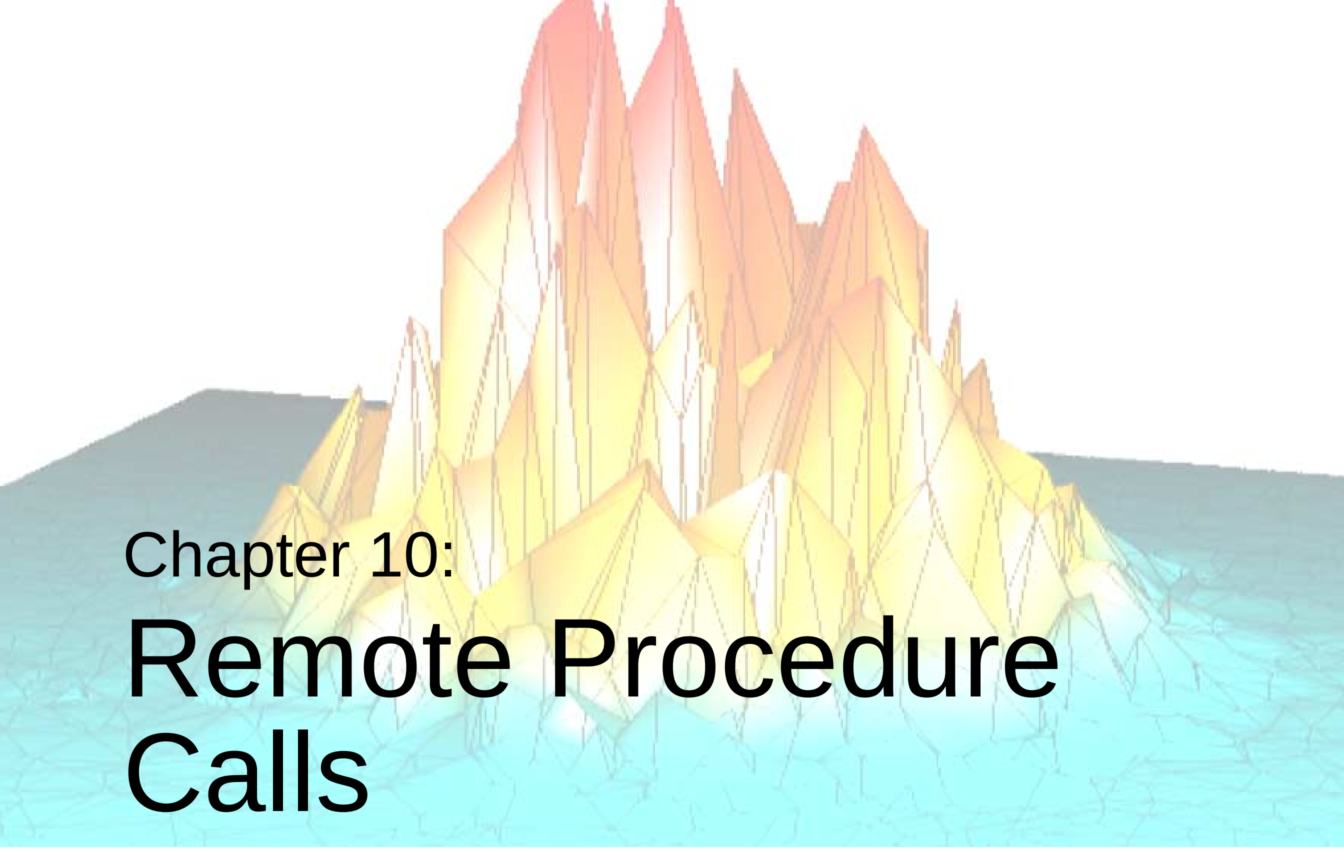
To call the example program from within IDL:

```
X = FINDGEN(10) ; Make an array.
sum = 0.0
S = CALL_EXTERNAL('example1.so', $
   'sum_array_', X, N_ELEMENTS(X), sum)
```

In this example, `example1.so` is the name of the sharable image file, `sum_array_` is the name of the entry point, and `X` and `N_ELEMENTS(X)` are passed to the called routine as parameters. The returned value is contained in the variable `sum.`

Note

The entry point name generated by the Fortran compiler may be different than that produced by the C compiler. One of the best ways to find out what name was generated is to use the UNIX `nm` utility on the object file. See your system's man page for `nm` for details.



Chapter 10: Remote Procedure Calls

This chapter discusses the following topics:

IDL and Remote Procedure Calls

Remote Procedure Calls (RPCs) allow one process (the *client* process) to have another process (the *server* process) execute a procedure call just as if the caller process had executed the procedure call in its own address space. Since the client and server are separate processes, they can reside on the same machine or on different machines. RPC libraries allow the creation of network applications without having to worry about underlying networking mechanisms.

IDL supports RPCs so that other applications can communicate with IDL. A library of C language routines is included to handle communication between client programs and the IDL server.

Note

Remote procedure calls are supported only on UNIX platforms.

The current implementation allows IDL to be run as an RPC server and your own program to be run as a client. IDL commands can be sent from your application to the IDL server, where they are executed. Variable structures can be defined in the client program and then sent to the IDL server for creation as IDL variables. Similarly, the values of variables in the IDL server session can be retrieved into the client process.

With the release of IDL version 5.0, IDL's RPC functionality has been completely revised and an new API created. The new RPC interface mirrors the API used by callable IDL. See "Compatibility with Older IDL Code" on page 227 for details.

Using IDL as an RPC Server

The IDL RPC Directory

All of the files related to using IDL's RPC capabilities are found in the `rpc` subdirectory of the `external` subdirectory of the main IDL directory. The main IDL directory is referred to here as *idldir*.

Running IDL in Server Mode

To use IDL as an RPC server, run IDL in server mode by using the `idlrpc` command. The RPC server can be invoked one of two ways:

```
idlrpc
```

or

```
idlrpc -server=server_number
```

where *server_number* is the hexadecimal server ID number (between 0x20000000 and 0x3FFFFFFF) for IDL to use. For example, to run IDL with the server ID number 0x20500000, use the command:

```
idlrpc -server=20500000
```

If a server ID number is not supplied, IDL uses the default, IDL_RPC_DEFAULT_ID, defined in the file *idldir*`/external/rpc/idl_rpc.h`. This value is originally set to 0x2010CAFE.

Client Variables

The IDL RPC client API uses the same data structure as IDL to represent a variable, namely an **IDL_VARIABLE** structure. By not using a unique data structure to represent a variable, the IDL RPC client API can follow a format that is similar to the API of Callable IDL.

When a variable is created by the IDL RPC client API (when a variable is returned from the **IDL_RPCGetMainVariable** function, for example) dynamic memory is allocated for the variable and for its value. These dynamic variables are similar to temporary variables which are used in IDL.

The IDL RPC client API provides routines to create, manipulate and delete dynamic or IDL RPC client temporary variables. These API routines follow the same format as the Callable IDL API and most have the same calling sequence.

When a client dynamic or temporary variable is no longer needed by the IDL RPC client program, use the **IDL_RPCDeltmp()** function to delete or free up the memory associated with the variable. Failure to delete a client temporary variable could result a memory “leak” in the client program.

Linking to the Client Library

To make use of the IDL RPC functionality, you will need to do the following:

- Include the file `idl_rpc.h` in your application.
- Have a copy of `idl_export.h` in the include path when you compile the client application.
- Link your client application to the IDL client shared object library (`libidl_rpc`).
- If the client library is linked as a shared object, you must set the shared object search path environment variable so that it includes the directory that contains the IDL client library.

  The name of this variable is normally LD_LIBRARY_PATH, except on HP and IBM systems, where the variable names are:
- HP: SHLIB_PATH
- IBM: LIBPATH

  If this variable is not set correctly, an error message will be issued by the system loader when the client program is started.

The command used to compile and link a client program to the IDL RPC client library follows the following format:

```
% cc -o example $(PRE_FLAGS) example.o -lidl_rpc
   $(POST_FLAGS)
```

where PRE_FLAGS and POST_FLAGS are platform dependent. The proper flags for each UNIX operating system supported by IDL are contained in the file `rpc_link.txt`, located in the in the `rpc` subdirectory of the `external` subdirectory of the main IDL directory.

Example of IDL RPC Client API

To use the IDL client side API, execute the following sequence of steps:

1. Call **IDL_RPCInit()** to connect to the server
2. Perform actions on the server—get and set variables, run IDL commands, etc.
3. Call **IDL_RPCCleanup()** to disconnect from the server.

The code shown in the following figure is an example that can be used to set up a remote session of IDL using the RPC features. Note that this C program will need to be linked against the supplied shared library `libidl_rpc`. This code is included in the *idldir*`/external/rpc` directory as `example.c`.

C

```
#include "idl_rpc.h"
int main()
{
     CLIENT *pClient;
     char    cmdBuffer[512];
     int     result;

 /* Connect to the server */
      if( (pClient = IDL_RPCInit(0, (char*)NULL)) == (CLIENT*)NULL){
          fprintf(stderr, "Can't register with IDL server\n");
          exit(1);
       }

/* Start a loop that will read commands and then send them to idl */
     for(;;){
         printf("RMTIDL> ");
         cmdBuffer[0]='\0';
         gets(cmdBuffer);
         if( cmdBuffer[0] == '\n' || cmdBuffer[0] == '\0')
              break;
         result = IDL_RPCExecuteStr(pClient, cmdBuffer);
                    }

 /* Now disconnect from the server and kill it. */
     if(!IDL_RPCCleanup(pClient, 1))
           fprintf(stderr, "IDL_RPCCleanup: failed\n");
     exit(0);
  }
```

Table 10-1: Remote Execution of IDL via RPC

Compile example.c with the appropriate flags for your platform, as described in "Linking to the Client Library" on page 225. Once this example is compiled, execute it using the following commands:

```
% idlrpc
```

Then, in another process:

```
% example
```

Compatibility with Older IDL Code

With the release of IDL 5.0, IDL's Remote Procedure Call functionality has been completely reworked. While RPC code built for older versions of IDL can still be used with IDL 5.0 and later, the new RPC functionality has the following advantages:

- The new API mirrors the Callable IDL API.
- The RPC client-side library is provided as a pre-built sharable library, eliminating the need to build the library on your system.
- The RPC server-side executable, `idlrpc`, is built using Callable IDL, providing an example of how Callable IDL can be used.
- Source code is provided for both the Server and Client side programs, allowing you to enhance IDL's RPC functionality.

RPC code built for versions of IDL prior to version 5.0 can be linked with IDL version 5 and later using a compatibility layer. This layer is contained in the files `idl_rpc_obsolete.c` and `idl_rpc_obsolete.h`.

To use the compatibility routines, include the file `lib_rpc_obsolete.h` in your application and use the following link statement as a template:

```
% cc -o old_example $(PRE_FLAGS) old_example.o \
idl_rpc_obsolete.o -lidl_rpc $(POST_FLAGS)
```

where the macros PRE_FLAGS and POST_FLAGS are the same as those described in "Linking to the Client Library" on page 225.

While the compatibility layer covers most of the old IDL RPC functionality, some of the more obscure operations have either been modified or are no longer supported. The features which have changed are as follows:

- **idl_server_interactive**: This function is no longer supported.
- **get_idl_variable**: The following return values are no longer supported:

Value	Description
`-2`	Illegal variable name (for example, “213xyz”, “#a”, “!DEVICE”)
`-3`	Variable not transportable (for example, the variable is a structure or associated variable)

Table 10-2: get_idl_variable Unsupported Values

- **set_idl_timeout**: the **tv_usec** field of the **timeval** struct is ignored.
- **idl_set_verbosity()**: This function is no longer supported.

All other functionality is supported.

The IDL RPC Library

The IDL RPC library contains several C language interface functions that facilitate communication between your application and IDL. There are functions to register and unregister clients, set timeouts, get and set the value of IDL variables, send commands to the IDL server, and cause the server to exit. These functions are:

- IDL_RPCCleanup
- IDL_RPCDeltmp
- IDL_RPCExecuteStr
- IDL_RPCGetMainVariable
- IDL_RPCGettmp
- IDL_RPCGetVariable
- IDL_RPCImportArray
- IDL_RPCInit
- IDL_RPCMakeArray
- IDL_RPCOutputCapture
- IDL_RPCOutputGetStr
- IDL_RPCSetMainVariable
- IDL_RPCSetVariable
- IDL_RPCStoreScalar
- IDL_RPCStrDelete
- IDL_RPCStrDup
- IDL_RPCStrEnsureLength
- IDL_RPCStrStore
- IDL_RPCTimeout
- IDL_RPCVarCopy
- IDL_RPCVarGetData
- Variable Accessor Macros

IDL_RPCCleanup

Calling Sequence

```
int IDL_RPCCleanup( CLIENT *pClient, int iKill)
```

Description

Use this function to release the resources associated with the given CLIENT structure or to kill the IDL RPC server.

Parameters

pClient

A pointer to the CLIENT structure for the client/server connection to be disconnected.

iKill

Set **iKill** to a non-zero value to kill the server when the connection is broken.

Return Value

This function returns 1 on success, or 0 on failure.

IDL_RPCDeltmp

Calling Sequence

```
void IDL_RPCDeltmp( IDL_VPTR vTmp)
```

Description

Use this function to de-allocate all dynamic memory associated with the **IDL_VPTR** that is passed into the function. Once this function returns, any dynamic portion of **vTmp** is deallocated and should not be referenced.

Parameters

vTmp

The variable that will be de-allocated.

Return Value

None.

IDL_RPCExecuteStr

Calling Sequence

```
int IDL_RPCExecuteStr(CLIENT *pClient, char * pCommand)
```

Description

Use this function to send IDL commands to the IDL RPC server. The command is executed just as if it had been entered from the IDL command line.

This function cannot be used to send multiple line commands and will return an error if a “$” is detected at the end of the command string. It will also return an error if “$” is the first character, since this would spawn an interactive process and hang the IDL RPC server.

Parameters

pClient

A pointer to the CLIENT structure that corresponds to the desired IDL session.

pCommand

A null-terminated IDL command string.

Return Value

This function returns the following values:

1 — Success.

0 — Invalid command string.

For all other errors, the value of !ERROR_STATE.CODE is returned. This number could be passed as an argument to the IDL function **STRMESSAGE()** to determine the exact cause of the error.

IDL_RPCGetMainVariable

Calling Sequence

```
IDL_VPTR IDL_RPCGetMainVariable(CLIENT *pClient, char *Name)
```

Description

Call this function to get the value of an IDL RPC server main level variable referenced by the name contained in **Name**. **IDL_RPCGetMainVariable** will then return a pointer to an **IDL_VARIABLE** structure that contains the value of the variable.

Parameters

pClient

A pointer to the CLIENT structure that corresponds to the desired IDL session.

Name

The name of the variable to find.

Return Value

On success, this function returns a pointer to an **IDL_VARIABLE** structure that contains the value of the desired IDL RPC main level variable. On failure this function returns NULL.

Note that the returned variable is marked as **temporary** and should be deleted when the variable is no longer needed. For more information on IDL RPC variables, see "Client Variables" on page 224.

IDL_RPCGettmp

Calling Sequence

```
IDL_VPTR IDL_RPCGettmp(void)
```

Description

Use this function to create an **IDL_VPTR** to a dynamically allocated **IDL_VARIABLE** structure. When you are finished with this variable, pass it to **IDL_RPCDeltmp()** to free any memory allocated by the variable.

Parameters

None.

Return Value

On success, this function returns an **IDL_VPTR**. On failure, it returns NULL.

IDL_RPCGetVariable

Calling Sequence

```
IDL_VPTR IDL_RPCGetVariable(CLIENT *pClient, char *Name)
```

Description

Use this function to get a pointer to an **IDL_VARIABLE** structure that contains the value of an IDL RPC server variable referenced by **Name**. The current scope of the IDL program is used to get the value of the variable.

Parameters

pClient

A pointer to the CLIENT structure that corresponds to the desired IDL session.

Name

The name of the variable to find.

Return Value

On success, this function returns a pointer to an **IDL_VARIABLE** structure that contains the value of the desired IDL RPC variable. On failure this function returns NULL.

Note that the returned variable is marked as **temporary** and should be deleted when the variable is no longer needed. For more information on IDL RPC variables, see "Client Variables" on page 224.

IDL_RPCImportArray

Calling Sequence

```
IDL_VPTR IDL_RPCImportArray(int n_dim, IDL_MEMINT dim[],
    int type, UCHAR *data, IDL_ARRAY_FREE_CB free_cb)
```

Description

Use this function to create an IDL array variable whose data the server supplies, rather than having the client API allocate the data space.

Parameters

n_dim

The number of dimensions in the array.

dim

An array of **IDL_MAX_ARRAY_DIM** elements, containing the size of each dimension.

type

The IDL type code describing the data. IDL type codes are discussed in "Type Codes" on page 258.

data

A pointer to your array data.

free_cb

If non-NULL, **free_cb** is a pointer to a function that will be called when the IDL RPC client routines frees the array. This feature gives the caller a sure way to know when the data is no longer referenced. Use the called function to perform any required cleanup, such as freeing dynamic memory or releasing shared or mapped memory.

Return Value

An **IDL_VPTR** that points to an **IDL_VARIABLE** structure containing a reference to the imported array. This function returns NULL if the operation was unsuccessful.

IDL_RPCInit

Calling Sequence

```
Client *IDL_RPCInit(long ServerId, char* pHostname)
```

Description

Use this function to initialize an IDL RPC client session.

The client program is registered as a client of the IDL RPC server. The server that the client is registered with depends on the values of the parameters passed to the function.

Parameters

ServerId

The ID number of the IDL server that the program is to be registered with. If this value is 0, the default server ID (0x2010CAFE) is used.

pHostname

This is the name of the machine where the IDL server is running. If this value is NULL or “”, the default, “localhost”, is used.

Return Value

A pointer to the new CLIENT structure is returned upon successful completion. This opaque data structure is then later used by the client program to perform operations with the server. This function returns NULL if the operation was unsuccessful.

IDL_RPCMakeArray

Calling Sequence

```
char * IDL_RPCMakeArray( int type,  int n_dim, IDL_MEMINT dim[],
    int init, IDL_VPTR *var)
```

Description

This function creates an IDL RPC client temporary array variable with a data area of the specified size.

Parameters

type

The IDL type code for the resulting array. IDL type codes are discussed in “Type Codes” on page 258.

n_dim

The number of array dimensions. The constant **IDL_MAX_ARRAY_DIM** defines the upper limit of this value.

dim

A C array of **IDL_MAX_ARRAY_DIM** elements containing the array dimensions. The number of dimensions in the array is given by the **n_dim** argument.

init

This parameter specifies the sort of initialization that should be applied to the resulting array. **init** must be one of the following:

- IDL_ARR_INI_NOP — No initialization is done. The data area of the array will contain whatever garbage was left behind from its previous use.
- IDL_ARR_INI_ZERO — The data area of the array is zeroed.

var

The address of an **IDL_VPTR** containing the address of the resulting IDL RPC client temporary variable.

Return Value

On success, this function returns a pointer to the data area of the allocated array. The value returned is the same as is contained in the **var->value.arr->data** field of the variable. On failure, it returns NULL.

As with variables returned from **IDL_RPCGettmp()**, the variable allocated via this function must be de-allocated using **IDL_RPCDeltmp()** when the variable is no longer needed.

IDL_RPCOutputCapture

Calling Sequence

```
int IDL_RPCOutputCapture( CLIENT *pClient, int n_lines)
```

Description

Use this routine to enable and disable capture of lines output from the IDL RPC server. Normally, IDL will write any output to the terminal on which the server was started. This function can be used to save this information so that the client program can request the lines sent to the output buffer.

Parameters

pClient

A pointer to the CLIENT structure that corresponds to the desired IDL session.

n_lines

If this value is less than or equal to zero, no output lines will be buffered in the IDL RPC server and output will be sent to the normal output device on the IDL RPC server. If the value of this parameter is greater than zero, the specified number of lines will be stored by the IDL RPC server.

Return Value

This function returns 1 on success, or 0 on failure.

IDL_RPCOutputGetStr

Calling Sequence

```
int IDL_RPCOutputGetStr(CLIENT *pClient,  IDL_RPC_LINE_S *pLine,
    int first)
```

Description

Use this function to get an output line from the line queue being maintained on the RPC server. The routine **IDL_RPCOutputCapture()** *must* have been called to initialize the output queue on the RPC server before this routine is called.

Parameters

pClient

A pointer to the CLIENT structure that corresponds to the desired IDL session.

pLine

A pointer to a valid **IDL_RPC_LINE_S** structure. The **buf** field of this structure will contain the output string returned from the IDL RPC server and the flags field will be set to one of the following (from `idl_export.h`):

- IDL_TOUT_F_STDERR — Send the text to **stderr** rather than **stdout**, if that distinction means anything to your output device.
- IDL_TOUT_F_NLPOST — After outputting the text, start a new output line. On a tty, this is equivalent to sending a new line (‘`\n`) character.

first

If **first** is set equal to a non-zero value, the first line is popped from the output buffer on the IDL RPC server (the output buffer is treated like a stack). If **first** is set equal to zero, the last line is de-queued from the output buffer (the output buffer is treated like a queue).

Return value

A true value (1) is returned upon success. A false value (0) is returned when there are no more lines available in the output buffer or when an RPC error is detected.

IDL_RPCSetMainVariable

Calling Sequence

```
int IDL_RPCSetMainVariable( CLIENT *pClient,  char *Name,
    IDL_VPTR pVar)
```

Description

Use this routine to assign a value to a main level IDL variable in the IDL RPC server session referred to by **pClient**. If the variable does not already exist, a new variable will be created.

Parameters

pClient

A pointer to the CLIENT structure that corresponds to the desired IDL session.

Name

A pointer to the null-terminated name of the variable, which must be in upper-case.

pVar

A pointer to an **IDL_VARIABLE** structure that contains the value that the IDL RPC main level variable referenced by **Name** should be set to. For more information on creating this variable, see "Client Variables" on page 224.

Return Value

This function returns 1 on success, or 0 on failure.

IDL_RPCSetVariable

Calling Sequence

```
int IDL_RPCSetVariable( CLIENT *pClient,  char *Name,
    IDL_VPTR pVar)
```

Description

Use this routine to assign a value to an IDL variable in the IDL RPC server session referred to by **pClient**. If the variable does not already exist, a new variable will be created. Unlike **IDL_RPCSetMainVariable()**, this routine sets the variable in the current IDL program scope.

Parameters

pClient

A pointer to the CLIENT structure that corresponds to the desired IDL session.

Name

A pointer to the null-terminated name of the variable, which must be in upper-case.

pVar

A pointer to an **IDL_VARIABLE** structure that contains the value that the IDL RPC variable referenced by **Name** should be set to. For more information on creating this variable, see “Client Variables” on page 224.

Return Value

This function returns 1 on success, or 0 on failure.

IDL_RPCStoreScalar

Calling Sequence

```
void IDL_RPCStoreScalar(IDL_VPTR dest,  int type,
    IDL_ALLTYPES *value)
```

Description

Use this function to store a scalar value into an **IDL_VARIABLE** structure. Before the scalar is stored, any dynamic part of the existing **IDL_VARIABLE** is de-allocated.

Parameters

dest

An **IDL_VPTR** to the **IDL_VARIABLE** in which the scalar should be stored.

type

The type code for the scalar value. IDL type codes are discussed in "Type Codes" on page 258.

value

The address of an **IDL_ALLTYPES** union that contains the value to store.

Return Value

None.

IDL_RPCStrDelete

Calling Sequence

```
void IDL_RPCStrDelete(IDL_STRING *str, IDL_MEMINT n)
```

Description

Use this function to delete a string. See the description of **IDL_StrDelete()** in "Deleting Strings" on page 331.

IDL_RPCStrDup

Calling Sequence

```
void IDL_RPCStrDup(IDL_STRING *str, IDL_MEMINT n)
```

Description

Use this function to duplicate a string. See the description of **IDL_StrDup()** in "Copying Strings" on page 330.

IDL_RPCStrEnsureLength

Calling Sequence

```
void IDL_RPCStrEnsureLength(IDL_STRING *s, int n)
```

Description

Use this function to check the length of a string. See the description of **IDL_StrEnsureLength()** in "Obtaining a String of a Given Length" on page 333.

IDL_RPCStrStore

Calling Sequence

```
void IDL_RPCStrStore( IDL_STRING *s, char *fs)
```

Description

Use this function to store a string. See description of **IDL_StrStore** in "Setting an IDL_STRING Value" on page 332.

IDL_RPCTimeout

Calling Sequence

```
int IDL_RPCTimeout(long lTimeOut)
```

Description

Use this function to set the timeout value used when the RPC client makes requests of the server.

Parameters

lTimeOut

A integer value, in seconds, specifying the timeout value that will be used in RPC operations.

Return Value

This function returns 1 on success, or 0 on failure.

IDL_RPCVarCopy

Calling Sequence

```
void IDL_RPCVarCopy(IDL_VPTR src, IDL_VPTR dst)
```

Description

Use this function to copy the contents of the **src** variable to the **dst** variable. Any dynamic memory associated with **dst** is de-allocated before the source data is copied. This function emulates the callable IDL function **IDL_VarCopy()**.

Parameters

src

The source variable to be copied. If this variable is marked as temporary (returned from **IDL_RPCGettmp()**, for example) the dynamic data will be moved rather than copied to the destination variable.

dst

The destination variable that **src** is copied to.

Return Value

None.

IDL_RPCVarGetData

Calling Sequence

```
void IDL_RPCVarGetData(IDL_VPTR v, IDL_MEMINT *n, char **pd,
    int ensure_simple)
```

Description

Use this function to obtain a pointer to a variable's data, and to determine how many data elements the variable contains.

Parameters

v

The variable for which data is desired.

n

The address of a variable that will contain the number of elements in **v**.

pd

The address of a variable that will contain a pointer to **v**'s data, cast to be a pointer to pointer to char (e.g. (char **) &myptr).

ensure_simple

If TRUE, this routine calls the **ENSURE_SIMPLE** macro on the argument **v** to screen out variables of the types it prevents. Otherwise, **EXCLUDE_FILE** is called, because file variables have no data area to return.

Return Value

On exit, **IDL_RPCVarGetData()** stores the data count and pointer into the variables pointed at by **n** and **pd**, respectively.

Variable Accessor Macros

The following macros can be used to get information on IDL RPC variables. These macros are defined in `idl_rpc.h`.

All of these macros accept a single argument, *v*, of type **IDL_VPTR**.

IDL_RPCGetArrayData(*v*)

This macro returns a pointer (*char**) to the data area of an array block.

IDL_RPCGetArrayDimensions(*v*)

This macro returns a C array which contains the array dimensions.

IDL_RPCGetArrayNumDims(*v*)

This macro returns the number of dimensions of the array.

IDL_RPCGetVarByte(*v*)

This macro returns the value of a 1-byte, unsigned *char* variable.

IDL_RPCGetVarComplex(*v*)

This macro returns the value (as a *struct*, not a pointer) of a complex variable.

IDL_RPCGetVarComplexR(*v*)

This macro returns the real field of a complex variable.

IDL_RPCGetVarComplexI(*v*)

This macro returns the imaginary field of a complex variable.

IDL_RPCGetVarDComplex(*v*)

This macro returns the value (as a *struct*, not a pointer) of a double precision, complex variable.

IDL_RPCGetVarDComplexR(*v*)

This macro returns the real field of a double-precision complex variable.

IDL_RPCGetVarDComplexI(*v*)

This macro returns the imaginary field of a double-precision complex variable.

IDL_RPCGetVarDouble(*v*)

This macro returns the value of a double-precision, floating-point variable.

IDL_RPCGetVarFloat(*v*)

This macro returns the value of a single-precision, floating-point variable.

IDL_RPCGetVarInt(*v*)

This macro returns the value of a 2-byte integer variable.

IDL_RPCGetVarLong(*v*)

This macro returns the value of a 4-byte integer variable.

IDL_RPCGetVarLong64(*v*)

This macro returns the value of a 8-byte integer variable.

IDL_RPCVarIsArray(*v*)

This macro returns non-zero if *v* is an array variable.

IDL_RPCGetVarString(*v*)

This macro returns the value of a string variable (as a *char**).

IDL_RPCGetVarType(*v*)

This macro returns the type code of the variable. IDL type codes are discussed in "Type Codes" on page 258.

IDL_RPCGetVarUInt(*v*)

This macro returns the value of an unsigned 2-byte integer variable.

IDLRPCGetVarULong(*v*)

This macro returns the value of an unsigned 4-byte integer variable.

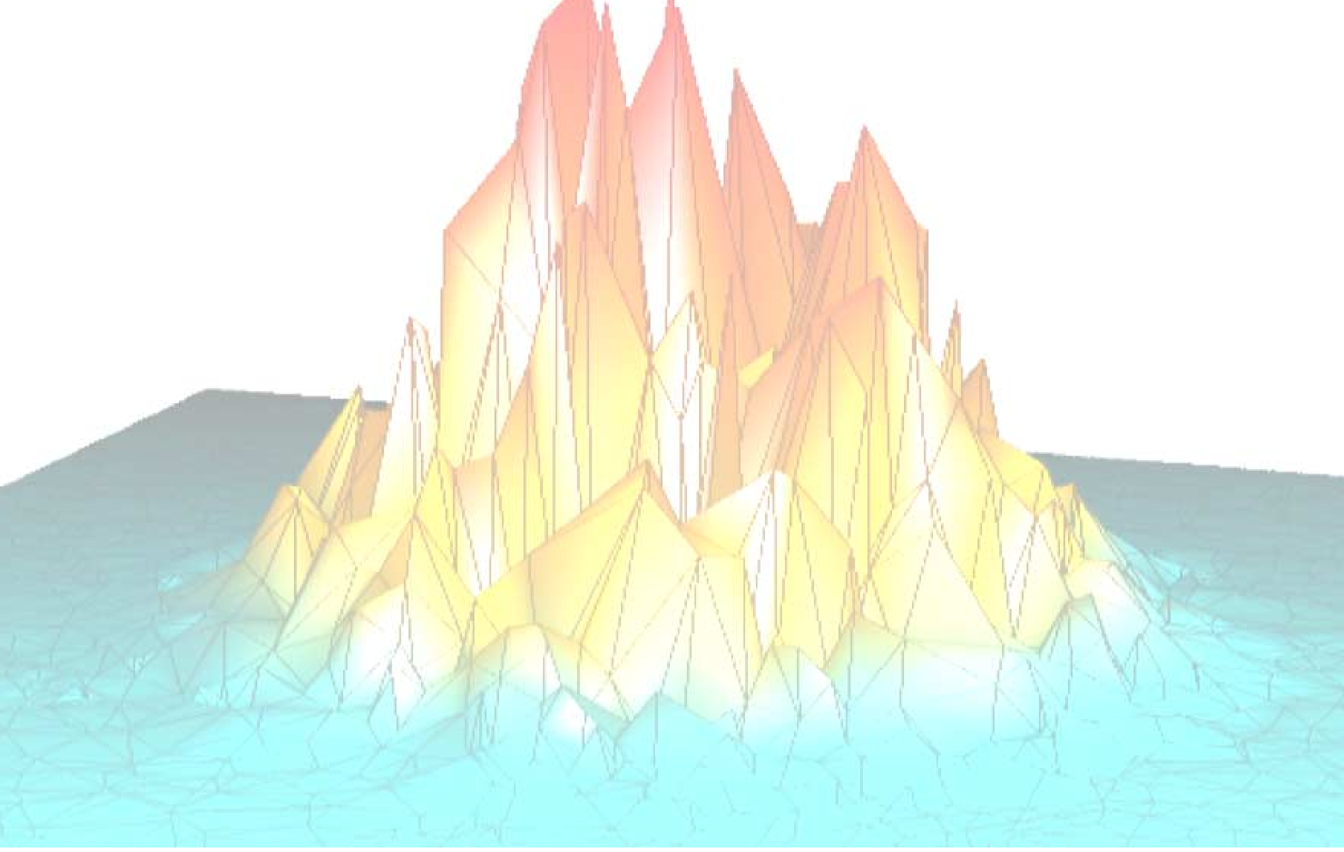
IDL_RPCGetVarULong64(*v*)

This macro returns the value of an unsigned 8-byte integer value.

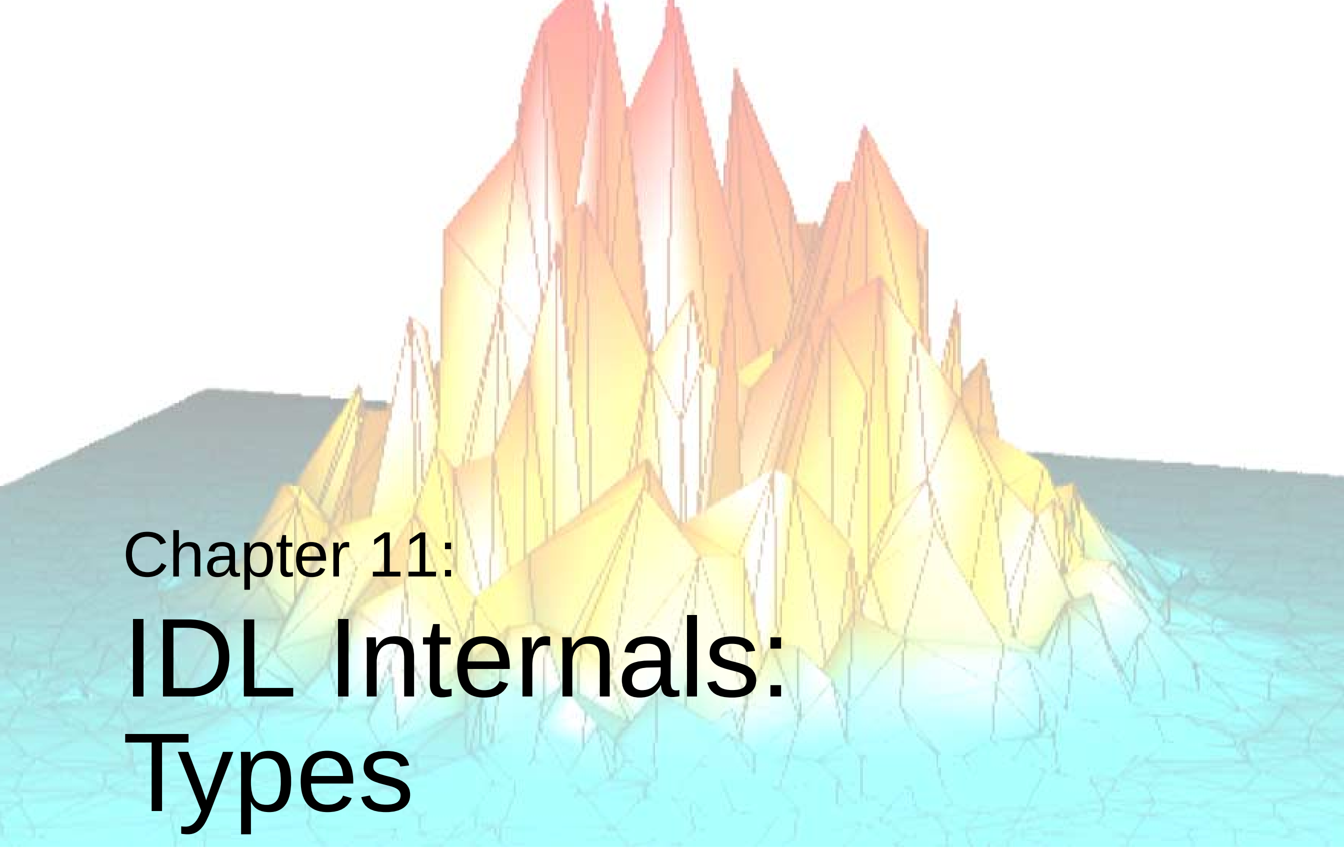
RPC Examples

A number of example files are included in the `*RSI_Directory*/external/rpc` directory. A `Makefile` for these examples is also included. These short C programs demonstrate the use of the IDL RPC library.

Source files for the `idlrpc` server program are located in the `*RSI_Directory*/external/rpc` directory. Note that you do not need to build the `idlrpc` server; it is pre-built and included in the IDL distribution. The `idlrpc` server source files are provided as examples only.



Part II: IDL’s Internal API



Chapter 11: IDL Internals: Types

This chapter describes the following topics:

Type Codes

Every IDL variable has a data type. The possible type codes and their mapping to C language types are listed in the following table. The undefined type code (**IDL_TYP_UNDEF**) will always have the value zero.

Although it is rare, the number of types could change someday. Therefore, you should always use the symbolic names when referring to any type except **IDL_TYP_UNDEF**. Even in the case of **IDL_TYP_UNDEF**, using the symbolic name will add clarity to your code. Note that all IDL structures are considered to be of a single type (**IDL_TYP_STRUCT**).

Clearly, distinctions must be made between various structures, but such distinctions are made at a different level. There are a few constants that can be used to make your code easier to read and less likely to break if/when the `idl_export.h` file changes. These are:

- **IDL_MAX_TYPE**—The value of the largest type.
- **IDL_NUM_TYPES**—The number of types. Since the types are numbered starting at zero, **IDL_NUM_TYPES** is one greater than **IDL_MAX_TYPE**.

Name	Type	C Type
IDL_TYP_UNDEF	Undefined	<None>
IDL_TYP_BYTE	Unsigned byte	UCHAR
IDL_TYP_INT	16–bit integer	IDL_INT
IDL_TYP_LONG	32–bit integer	IDL_LONG
IDL_TYP_FLOAT	Single precision floating	float
IDL_TYP_DOUBLE	Double precision floating	double
IDL_TYP_COMPLEX	Single precision complex	IDL_COMPLEX
IDL_TYP_STRING	String	IDL_STRING
IDL_TYP_STRUCT	Structure	See “Structure Variables” on page 273
IDL_TYP_DCOMPLEX	Double precision complex	IDL_DCOMPLEX

Table 11-1: IDL Types and Mapping to C

Name	Type	C Type
IDL_TYP_PTR	32–bit integer	IDL_ULONG
IDL_TYP_OBJREF	32–bit integer	IDL_ULONG
IDL_TYP_UINT	Unsigned 16-bit integer	IDL_UINT
IDL_TYP_ULONG	Unsigned 32-bit integer	IDL_ULONG
IDL_TYP_LONG64	64-bit integer	IDL_LONG64
IDL_TYP_ULONG64	Unsigned 64-bit integer	IDL_ULONG64

Table 11-1: (Continued) IDL Types and Mapping to C (Continued)

Type Masks

There are some situations in which it is necessary to specify types in the form of a bit mask rather than the usual type codes, for example when a single argument to a function can represent more than a single type. For any given type, the bit mask value can be computed as:

$$\mathrm{Mask} = 2^{\mathrm{TypeCode}}$$

The **IDL_TYP_MASK** preprocessor macro is provided to calculate these masks. Given a type code, it returns the bit mask. For example, to specify a bit mask for all the integer types:

```
IDL_TYP_MASK(IDL_TYP_BYTE)|IDL_TYP_MASK(IDL_TYP_INT)|
                           IDL_TYP_MASK(IDL_TYP_LONG)
```

Specifying all the possible types would require a long statement similar to the one above. To avoid having to type so much for this common case, the **IDL_TYP_B_ALL** constant is provided.

Mapping of Basic Types

Within IDL, the IDL data types are mapped into data types supported by the C language. Most of the types map directly into C primitives, while **IDL_TYP_COMPLEX**, **IDL_TYP_DCOMPLEX**, and **IDL_TYP_STRING** are defined as C structures. The mappings are given in the following table. Structures are built out of the basic types by laying them out in memory in the specified order using the same alignment rules used by the C compiler for the target machine.

Unsigned Byte Data

UCHAR is defined to be unsigned char in `idl_export.h`.

Integer Data

IDL_INT represents the signed 16-bit data type and is defined in `idl_export.h.`

Unsigned Integer Data

IDL_UINT represents the unsigned 16-bit data type and is defined in `idl_export.h.`

Long Integer Data

IDL long integers are defined to be 32-bits in size. The C long data type is not correct on all systems because C compilers for 64-bit architectures usually define long as 64-bits. Hence, the **IDL_LONG** typedef, declared in `idl_export.h` is used instead.

Unsigned Long Integer Data

IDL_ULONG represents the unsigned 32-bit data type and is defined in `idl_export.h`.

64-bit Integer Data

IDL_LONG64 represents the 64-bit data type and is defined in `idl_export.h`.

Unsigned 64-bit Integer Data

IDL_ULONG64 represents the unsigned 64-bit data type and is defined in `idl_export.h`.

Complex Data

The **IDL_TYP_COMPLEX** and **IDL_TYP_DCOMPLEX** data types are defined by the following C declarations:

```
typedef struct { float r, i; } IDL_COMPLEX;
typedef struct { double r, i; } IDL_DCOMPLEX;
```

This is the same mapping used by Fortran compilers to implement their complex data types, which allows sharing binary data with such programs.

String Data

The **IDL_TYP_STRING** data type is implemented by a string descriptor:

```
typedef struct {
  IDL_STRING_SLEN_T slen;  /* Length of string */
  short stype;           /* Type of string */
  char *s;               /* Pointer to string */
} IDL_STRING;
```

The fields of the **IDL_STRING** struct are defined as follows:

slen

The length of the string, not counting the null termination. For example, the string “Hello” has 5 characters.

stype

If **stype** is zero, the string pointed at by **s** (if any) was not allocated from dynamic memory, and should not be freed. If non-zero, **s** points at a string allocated from dynamic memory, and should be freed before being replaced. For information on dynamic memory, see “Dynamic Memory” on page 396 and “Getting Dynamic Memory” on page 288.

s

If **slen** is non-zero, **s** is a pointer to a null-terminated string of **slen** characters. If **slen** is zero, **s** should not be used. The use of a string pointer to memory located outside the **IDL_STRING** structure itself allows IDL strings to have dynamically-variable lengths.

Note

Strings are the most complicated basic data type, and as such, are at the root of more coding errors than the other types. See "IDL Internals: String Processing" on page 327.

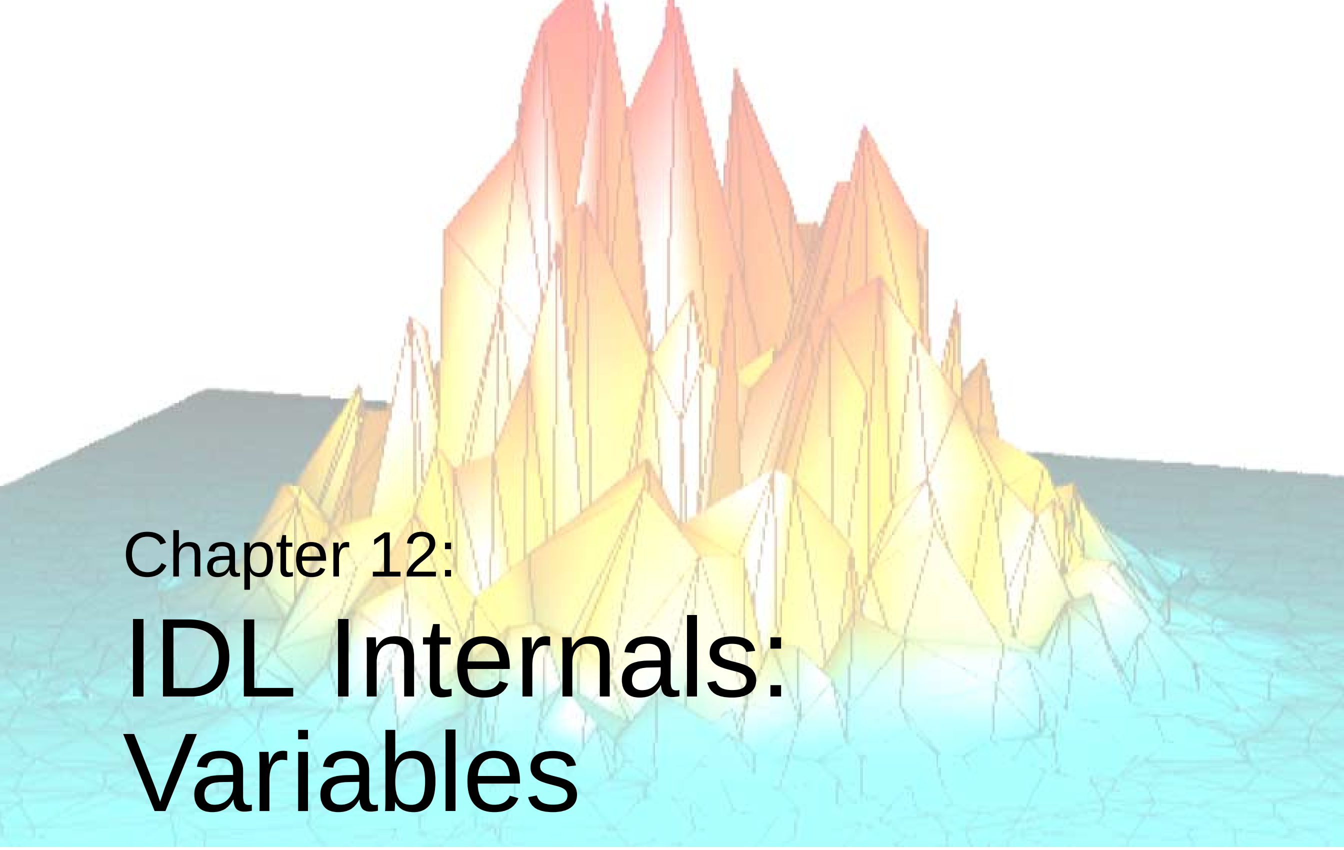
IDL_MEMINT and IDL_FILEINT Types

Some of the IDL-supported operating systems limit memory and file lengths to a signed 32-bit integer (approximately 2.3 GB). Some systems have 64-bit memory capabilities and others allow files longer than 2^{31}-1 bytes despite being 32-bit memory limited. To gracefully handle these differences without using conditional code, IDL internals use two special types, IDL_TYP_MEMINT (data type IDL_MEMINT) and IDL_TYP_FILEINT (data type IDL_FILEINT) to represent memory and file length limits.

IDL_MEMINT and IDL_FILEINT are not separate and distinct types; they are actually mappings to the IDL types discussed in "Mapping of Basic Types" on page 260. Specifically, they will be IDL_LONG for 32-bit quantities, and IDL_LONG64 for 64-bit quantities.

As an IDL internals programmer, you should not write code that depends on the actual machine type represented by these abstract types. To ensure that your code runs properly on all systems, use IDL_MEMINT and IDL_FILEINT in place of more specific types. These types can be used anywhere that a normal IDL type can be used, such as in keyword processing. Their systematic use for these purposes will ensure that your code is correct on any IDL platform.

Programmers should be aware of the IDL_MEMINTScalar() and IDL_FILEINTScalar() functions, described in "Converting Arguments to C Scalars" on page 350.



Chapter 12: IDL Internals: Variables

This chapter discusses the following topics:

IDL and Internal Variables

This chapter describes how variables are created and managed within IDL. While reading this chapter, you should refer to the following figure to see how each part fits into the overall structure of an IDL variable.

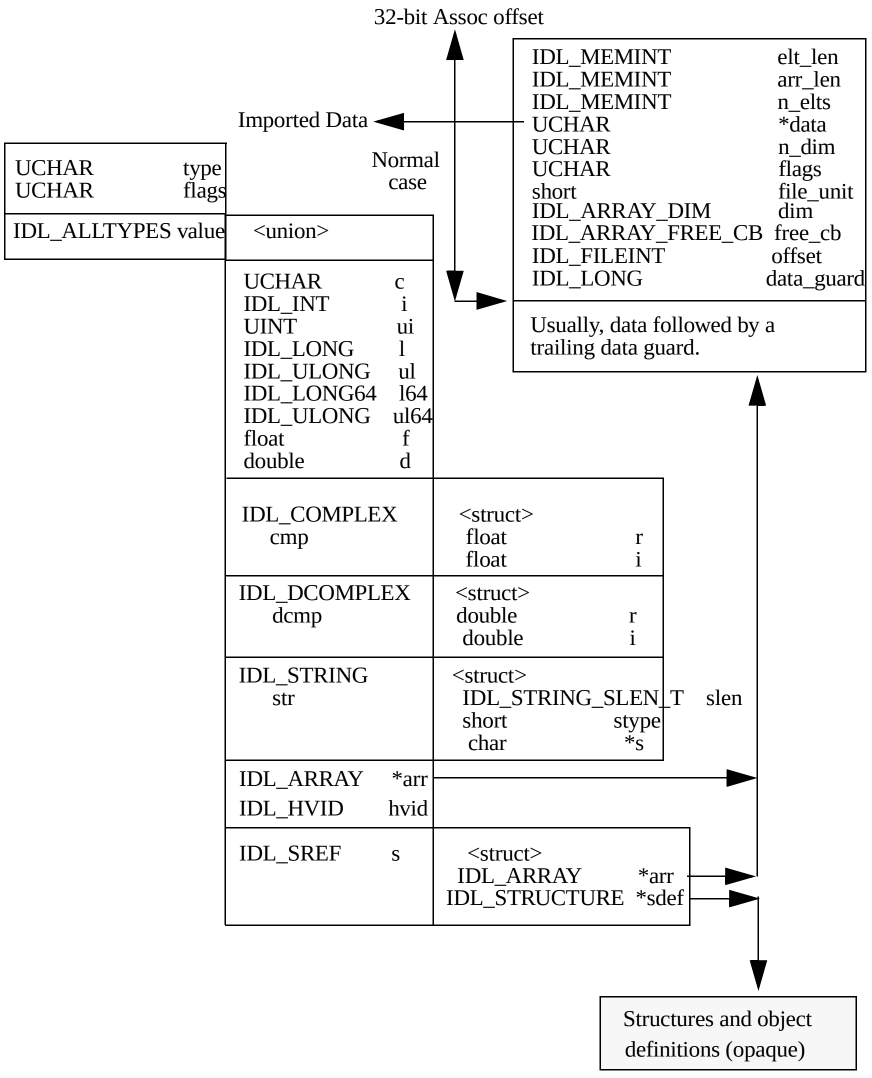


Figure 12-1: Structure of an IDL variable

The IDL_VARIABLE Structure

IDL variables are represented by **IDL_VARIABLE** structures. The definition of **IDL_VARIABLE** is as follows:

```
typedef struct {
  UCHAR type;
  UCHAR flags;
  IDL_ALLTYPES value;
}  IDL_VARIABLE;
```

An **IDL_VPTR** is a pointer to an **IDL_VARIABLE** structure:

```
typedef IDL_VARIABLE *IDL_VPTR;
```

The **IDL_ALLTYPES** union is defined as:

```
typedef union {
  UCHAR c;           /* Scalar IDL_TYP_BYTE */
  IDL_INT i;         /* Scalar IDL_TYP_INT */
  IDL_UINT ui;       /* Unsigned short integer value */
  IDL_LONG l;        /* Scalar IDL_TYP_LONG */
  IDL_ULONG ul;      /* Unsigned long value */
  IDL_LONG64 l64;    /* 64-bit integer value */
  IDL_ULONG64 ul64;  /* Unsigned 64-bit integer value */
  float f;           /* Scalar IDL_TYP_FLOAT */
  double d;          /* Scalar IDL_TYP_DOUBLE */
  IDL_COMPLEX cmp;   /* Scalar IDL_TYP_COMPLEX */
  IDL_DCOMPLEX dcmp; /* Scalar IDL_TYP_DCOMPLEX */
  IDL_STRING str;    /* Scalar IDL_TYP_STRING */
  IDL_ARRAY *arr;    /* Pointer to array descriptor */
  IDL_SREF s;        /* Structure descriptor */
  IDL_HVID hvid;     /* Heap variable identifier */
}IDL_ALLTYPES;
```

The basic scalar types are contained directly in this union, while arrays and structures are represented by the **IDL_ARRAY** and **IDL_SREF** structures that are discussed later in this chapter. The type field of the **IDL_VARIABLE** structure contains one of the type codes discussed in "Type Codes" on page 258. When a variable is initially created, it is given the type code **IDL_TYP_UNDEF**, indicating that the variable contains no value.

The **flags** field is a bit mask that specifies information about the variable. As a programmer adding code to the IDL system, you will rarely need to set bits in this mask. These bits are set by whatever portion of IDL created the variable. You can check them to make sure the characteristics of the variable fit the requirements of your routine (see "Checking Arguments" on page 345). The defined bits in the mask are:

IDL_V_CONST

If this flag is set, the variable is actually a constant. This means that storage for the **IDL_VARIABLE** resides inside the code section of the user procedure or function that used it. The IDL compiler generates such **IDL_VARIABLE**s when an expression involving a constant occurs. For example, the IDL statement:

```
PRINT, 23 * A
```

causes the compiler to generate a constant for the “23”. You must not change the value of this type of “variable”.

IDL_V_TEMP

If this flag is set, the variable is a temporary variable. IDL maintains a pool of nameless **IDL_VARIABLE**s that can be checked out and returned as needed. Such variables are used by the interpreter to temporarily store the results of expressions on the stack. For example, the statement:

```
PRINT, 2 * 3
```

will cause the interpreter to go through a sequence of events similar to:

1. Push a constant variable for the 2 on the stack.
2. Push a constant variable for the 3 on the stack.
3. Allocate a temporary variable, pop the two constants from the stack, perform the multiplication with the result going into the temporary variable.
4. Push the temporary variable onto the stack.
5. Call the **PRINT** system procedure specifying one argument.
6. Remove the argument to **PRINT** from the stack, and return the temporary variable.

Temporary variables are also used inside user procedures and functions. See “Temporary Variables” on page 279.

IDL_V_ARR

If this flag is set, the variable is an array, and the value field of the IDL_VARIABLE points to an array descriptor.

IDL_V_FILE

If this flag is set, the variable is a file variable, as created by IDL’s ASSOC function.

IDL_V_DYNAMIC

If this flag is set, the memory used by this **IDL_VARIABLE** is dynamically allocated. This bit is set for arrays, structures, and for variables of **IDL_TYP_STRING** (because the memory referenced via the string pointer is dynamic).

IDL_V_STRUCT

If this flag is set, the variable is a structure, and the value field of the **IDL_VARIABLE** points to the structure descriptor. For implementation reasons, all structure variables are also arrays, so **IDL_V_STRUCT** also implies **IDL_V_ARR**. Therefore, it is impossible to have a scalar structure. However, single-element structure arrays are quite common.

Because structure variables have their type field set to **IDL_TYP_STRUCT**, the **IDL_V_STRUCT** bit is redundant. It exists for efficiency reasons.

Scalar Variables

A scalar **IDL_VARIABLE** is distinguished by not having the **IDL_V_ARR** bit set in its **flags** field. A scalar variable must have one of the basic data types (IDL structures are never scalar) shown in Table 12-1. The data for a scalar variable is stored in the **IDL_VARIABLE** itself, using the **IDL_ALLTYPES** union. The following table gives the relationship between the data type and the field used.

Scalar Data Type	Field that Stores Data
IDL_TYP_UNDEF	None.
IDL_TYP_BYTE	value.c
IDL_TYP_INT	value.i
IDL_TYP_UINT	value.ui
IDL_TYP_LONG	value.l
IDL_TYP_ULONG	value.ul
IDL_TYP_LONG64	value.l64
IDL_TYP_ULONG64	value.ul64
IDL_TYP_FLOAT	value.f
IDL_TYP_DOUBLE	value.d
IDL_TYP_COMPLEX	value.cmp
IDL_TYP_DCOMPLEX	value.dcmp
IDL_TYP_STRING	value.str
IDL_TYP_PTR	value.hvid
IDL_TYP_OBJ	value.hvid

Table 12-1: Scalar Variable Data Locations

Array Variables

Array variables have the IDL_V_ARR bit of their **flags** field set, and the **value.arr** field points to an array descriptor defined by the **IDL_ARRAY** structure:

```
typedef IDL_MEMINT IDL_ARRAY_DIM[IDL_MAX_ARRAY_DIM];

typedef struct {
  IDL_MEMINT elt_len;
  IDL_MEMINT arr_len;
  IDL_MEMINT n_elts;
  UCHAR *data;
  UCHAR n_dim;
  UCHAR flags;
  short file_unit;
  IDL_ARRAY_DIM dim;
} IDL_ARRAY;
```

The meaning of the fields of an array descriptor are:

elt_len

The length of each array element in bytes (chars). The array descriptor does not keep track of the types of the array elements, only their lengths. Single elements can get quite long in the case of structures.

For IDL structures, this value includes any padding necessary to properly align the data along required boundaries. On a given platform, IDL creates structures the same way a C compiler does on that platform. As a result, you should not assume that the size of a structure is the sum of the sizes of the structure fields, or that the field offsets are in specific locations.

arr_len

The length of the entire array in bytes. This value could be calculated as (**elt_len** * **n_elts**), but is used so frequently that it is maintained as a separate field in the **IDL_ARRAY** struct.

n_elts

The number of elements in the array.

data

A pointer to the data area for the array. This is a region of dynamically allocated memory **arr_len** bytes long. This pointer should be cast to be a pointer of the correct type for the data being manipulated. For example, if the array variable being processed is pointed at by an **IDL_VPTR** named **v** and contains **IDL_TYP_INT** data:

```
IDL_INT *data;      /* Declare a pointer variable */
data = (IDL_INT *) v->value.arr->data;
```

n_dim

The number of array dimensions. The constant **IDL_MAX_ARRAY_DIM** defines the upper limit of this value.

flags

A bit mask that specifies characteristics of the array. Allowed values are:

IDL_A_FILE — This flag indicates that the array is a file variable, as created by the ASSOC function. The variable has an array block to specify the structure of the variable, but it has no data area. The data field of the IDL_ARRAY structure does not contain useful information, and should not be used.

IDL_A_PACKED — If array is an IDL_A_FILE variable and the data type is IDL_TYP_STRUCT, then Input/Output to this struct should use a packed data layout compatible with WRITEU instead of being a direct mapping onto the struct (which reflects the C compiler layout of the structure including its alignment holes).

file_unit

When the **IDL_A_FILE** bit is set in the **flags** field, **file_unit** contains the IDL Logical Unit Number associated with the variable.

dim

An array that contains the dimensions of the IDL variable. There can be up to **IDL_MAX_ARRAY_DIM** dimensions. The *number* of dimensions in the current array is given by the **n_dim** field.

Structure Variables

Structure variables have the type code **IDL_TYP_STRUCT**. They also have the **IDL_V_STRUCT** bit set in their **flags** field. The **value.s** field of such a variable contains a structure descriptor defined by the **IDL_SREF** structure:

```
typedef struct {
  IDL_ARRAY *arr;     /* ^ to IDL_ARRAY containing data */
  void *sdef;         /* ^ to structure definition */
} IDL_SREF;
```

The **arr** field points at an array block, as described on page 271. It is worth noting that in the definition of the **IDL_ALLTYPES** union (described on page 267), the **arr** field is a pointer to **IDL_ARRAY**, while the **s** field is an **IDL_SREF**, a structure that contains a pointer to **IDL_ARRAY** as its first member.

The resulting definition looks like:

```
union {
  IDL_ARRAY arr;
  struct {
    IDL_ARRAY arr;
    void *sdef;
  } s;
} value;
```

Due to the way C lays out fields in structs and unions, **value.arr** will have the same offset and size within the value union as **value.s.arr**. Therefore, it is possible to access the array block of a structure variable as **var->value.arr** rather than the more correct **var->value.s.arr**. You should avoid use of this shorthand, however, because it is not strictly correct usage and because RSI reserves the right to change the **IDL_SREF** definition in a way that could cause the memory layout of the ALLTYPES union to change.

Creating Structures

The actual structure definition is accessed through the **sdef** field, which is a pointer to an opaque IDL structure definition. Although the implementation of structure definitions is not public information, they can be created using the **IDL_MakeStruct()** function from a structure name and a list of tags:

```
void *IDL_MakeStruct(char *name, IDL_STRUCT_TAG_DEF *tags)
```

name

The name of the structure definition, or NULL for anonymous structures.

tags

An array of **IDL_STRUCT_TAG_DEF** elements, one for each tag.

The result from this function can be passed to **IDL_ImportArray()** or **IDL_ImportNamedArray()**, as described on page 286.

IDL_STRUCT_TAG_DEF is defined as:

```
typedef struct {
  char *name;
  IDL_MEMINT *dims;
  void *type;
  UCHAR flags;
} IDL_STRUCT_TAG_DEF;
```

name

Null-terminated uppercase name of the tag.

dims

An array that contains information about the dimensions of the structure. The first element of this array is the number of dimensions. Following elements contain the size of each dimension.

type

Either a pointer to another structure definition, or a simple IDL type code cast to void (e.g., **(void *) IDL_TYP_BYTE**).

flags

A bit mask that specifies additional characteristics of the tag. Allowed values are:

IDL_STD_INHERIT — Type must be IDL_TYP_STRUCT. This flag indicates that the structure is inherited (inlined) instead of making it a sub-structure as usual.

The following example shows how to define an anonymous structure. Suppose that you want to create a structure whose definition in the IDL language is:

```
{TAG1: 0L, TAG2: FLTARR(2,3,4), TAG3: STRARR(10)}
```

It can be created with **IDL_MakeStruct()** as follows:

```
static IDL_MEMINT one = 1;
static IDL_MEMINT tag2_dims[] = { 3, 2, 3, 4};
static IDL_MEMINT tag3_dims[] = { 1, 10 };
static IDL_STRUCT_TAG_DEF s_tags[] = {
  { "TAG1", 0, (void *) IDL_TYP_LONG},
  { "TAG2", tag2_dims, (void *) IDL_TYP_FLOAT},
  { "TAG3", tag3_dims, (void *) IDL_TYP_STRING},
  { 0 }
};
typedef struct data_struct {
  IDL_LONG tag1_data;
  float tag2_data [4] [3] [2];
  IDL_STRING tag_3_data [10];
} DATA_STRUCT;
static DATA_STRUCT s_data;
void *s;
IDL_VPTR v;

/* Create the structure definition */
s = IDL_MakeStruct(0, s_tags);
/* Import the data area s_data into an IDL structure,
   note that no data are moved. */
v = IDL_ImportArray(1, &one, IDL_TYP_STRUCT,
                (UCHAR *) &s_data, 0, s);
```

Accessing Structure Tags

Given an opaque IDL structure definition, you can determine the offset of the data and a description of its size and form (scalar, array, etc) for a given tag. **IDL_StructTagInfoByName()** returns this information given the name of the tag. **IDL_StructTagInfoByIndex()** does the same thing, given the numeric index of the tag. They are essentially the same routine, although **IDL_StructTagInfoByIndex()** is slightly more efficient:

```
IDL_MEMINT IDL_StructTagInfoByName(IDL_StructDefPtr sdef,
                                   char *name, int msg_action,
                                   IDL_VPTR *var)
IDL_MEMINT IDL_StructTagInfoByIndex(IDL_StructDefPtr sdef,
                                    int index,int msg_action,
                                    IDL_VPTR *var)
```

where:

sdef

Structure definition for which offset is needed.

name (IDL_StructTagInfoByName)

Name of tag for which information is required.

index (IDL_StructTagInfoByIndex)

Zero based index of tag for which information is required.

msg_action

The parameter that will be passed directly to **IDL_Message()** if the specified tag cannot be found in the supplied structure definition.

var

NULL, or the address of an **IDL_VPTR** to be filled in with a pointer to the variable description for the specified field.

On success, these functions return the data offset of the tag. On error, if the resulting call to **IDL_Message()** returns to the caller, a -1 is returned. The data offset can be added to the data pointer of an IDL variable of this structure type to obtain a pointer to the actual data for that tag.

If the tag is successfully located and the var argument is non-NULL, the **IDL_VPTR** it points at is filled in with a pointer to an **IDL_VARIABLE** structure that describes the type and organization of the tag. It is important to understand that this **IDL_VARIABLE** does not contain any actual data (or in the case of an array tag, a valid data pointer). Hence, the data part of the **IDL_VARIABLE** description should be ignored.

Determining the Number Of Structure Tags

One often needs to know how many tags a structure definition has in order to make use of the information supplied by the routines described above. The **IDL_StructNumTags()** function returns this information:

```
int IDL_StructNumTags(IDL_StructDefPtr sdef)
```

where:

sdef

Structure definition for which offset is needed.

Determining the Names Of Structures and their Tags

The **IDL_StructTagNameByIndex()** function returns the name of a specified tag from a structure definition, and optionally the name of the structure:

```
char *IDL_StructTagNameByIndex(IDL_StructDefPtr sdef, int index,
                int msg_action, char **struct_name)
```

where:

sdef

Structure definition for which name information is needed.

index

Zero based index of tag within the structure.

msg_action

The parameter that will be passed directly to IDL_Message() if the specified tag cannot be found in the supplied structure definition.

struct_name

NULL, or the address of a character pointer (char *) to be filled in with a pointer to the name of the structure. If the structure is anonymous, the string `“<Anonymous>”` is returned.

On success, a pointer to the tag name is returned. On error, if the resulting call to **IDL_Message()** returns to the caller, a NULL pointer is returned.

All strings returned by this function must be considered read-only, and must not be modified by the caller.

Heap Variables

Direct access to pointer and object reference heap variables (types **IDL_TYP_PTR** and **IDL_TYP_OBJREF**, respectively) is not allowed. Rather than accessing the heap variable directly, store the value of the heap variable (an IDL pointer or object reference) in a regular IDL variable at the IDL user level. Access the data in the regular variable, then store the results back in the heap variable (via the pointer or object reference) when done.

Note

You can use IDL’s TEMPORARY function to avoid making copies of the data.

Temporary Variables

As discussed previously, IDL maintains a pool of nameless variables known as temporary variables. These variables are used by the interpreter to hold temporary results from evaluating expressions, and are also used within system procedures and functions that need temporary workspace. In addition, system functions often obtain a temporary variable to return the result of their operation to the interpreter. Temporary variables have the following characteristics:

- All temporaries, when initially allocated, are of type **IDL_TYP_UNDEF**.
- Temporary variables do not have a name associated with them.
- Routines that check out temporaries must either check them back in or return them as the result of the function. Once you return a temporary variable, you cannot access it again.
- Temporary variables are reclaimed by the interpreter when it is about to exit after executing a program, so it is not possible to lose them and leak dynamic memory by allocating them and failing to return them. If the interpreter is exiting normally and it detects temporaries that have not been returned, it issues an error message. Such an error message indicates an error in the implementation of your system routine. If your routine exits by issuing an IDL_MSG_LONGJMP or IDL_MSG_IO_LONGJMP error via IDL_Message() however, allocated temporaries are expected, and are reclaimed quietly. Hence, your routines need only return temporaries on normal return, and not before issuing errors. See "IDL Internals: Error Handling" on page 335.

The interpreter uses temporary variables to hold values that are the result of evaluating expressions. Such temporaries are pushed on the interpreter stack where they are often passed as arguments to other routines. For example, the IDL statement:

```
PRINT, MAX(FINDGEN(100))
```

causes the interpreter to perform the following steps:

1. Push a constant variable with the value 100 onto the stack.
2. Call the system function FINDGEN, passing it one argument.
3. FINDGEN returns a temporary variable which is a 100-element vector with each element set to the value of its index.
4. The interpreter removes the arguments to FINDGEN from the stack (the constant 100) and pushes the resulting temporary variable onto the stack.

5. The MAX system function is called with a single argument—the temporary result from FINDGEN.
6. MAX finds the largest element in its argument (99), places that value into a temporary scalar variable, and returns that temporary variable as its result.
7. The interpreter removes the argument to MAX from the stack. This was the temporary array from FINDGEN, so it is returned to the pool of temporary variables. The resulting temporary variable from MAX is then pushed onto the stack.
8. The PRINT system procedure is called with a single argument, which is the temporary scalar variable from MAX. It prints the value of the variable and returns.
9. The interpreter removes the argument to PRINT from the stack, and returns it to the pool of temporary variables.

Getting a Temporary Variable

Temporary variables are obtained via the **IDL_Gettmp()** function:

```
IDL_VPTR IDL_Gettmp(void);
```

IDL_Gettmp() requires no arguments, and returns an IDL_VPTR to a temporary variable. This variable must be returned to the pool of temporary variables (with a call to **IDL_Deltmp()**) or be returned as the value of a system function before control returns to the interpreter, or an error will occur.

A number of variants on **IDL_Gettmp()** exist, as convenience routines for creating temporary scalar variables of a given type and value. In all cases, the value is supplied as the sole argument, and the resulting type is indicated by the name of the routine:

```
IDL_VPTR IDL_GettmpInt(IDL_INT value);
IDL_VPTR IDL_GettmpUInt(IDL_UINT value);
IDL_VPTR IDL_GettmpLong(IDL_LONG value);
IDL_VPTR IDL_GettmpULong(IDL_ULONG value);
IDL_VPTR IDL_GettmpFILEINT(IDL_FILEINT value);
IDL_VPTR IDL_GettmpMEMINT(IDL_MEMINT value);
```

Creating a Temporary Array

Temporary array variables can be obtained via the **IDL_MakeTempArray()** function:

```
char *IDL_MakeTempArray(int type, int n_dim, IDL_MEMINT dim[],
                        int init, IDL_VPTR *var)
```

where:

type

The type code for the resulting array. See “Type Codes” on page 258.

n_dim

The number of array dimensions. The constant **IDL_MAX_ARRAY_DIM** defines the upper limit of this value.

dim

An array of **IDL_MAX_ARRAY_DIM** elements containing the array dimensions. The number of dimensions in the array is given by the **n_dim** argument.

init

Specifies the sort of initialization that should be applied to the resulting array. The **init** argument must be one of the following:

- **IDL_ARR_INI_INDEX** — Each element of the array is set to the value of its index. The INDGEN family of built-in system functions is implemented using this feature.
- **IDL_ARR_INI_NOP** — No initialization is done. The data area of the array will contain whatever garbage was left behind from its previous use. Experience has shown that **IDL_TYP_STRING** data should never be left uninitialized due to the risk of dereferencing an invalid string pointer and crashing IDL. Therefore, **IDL_TYP_STRING** data is zeroed when **IDL_ARR_INI_NOP** is specified.
- **IDL_ARR_INI_ZERO** — The data area of the array is zeroed.

var

The address of an **IDL_VPTR** where the address of the resulting temporary variable will be put.

The data area of an array **IDL_VARIABLE** is accessible from its **IDL_VPTR** as **var->value.arr->data**. However, since most routines that create an array need to access the data area, **IDL_MakeTempArray()** returns the data area pointer as its value. As with **IDL_Gettmp()**, the variable allocated via **IDL_MakeTempArray()**

must be returned to the pool of temporary variables or be returned as the value of a system function before control returns to the interpreter, or an error will occur.

Creating a Temporary Vector

IDL_MakeTempArray() can be used to create arrays with any number of dimensions, but the common case of creating a 1-dimensional vector can be carried out more conveniently using the **IDL_MakeTempVector()** function:

```
char *IDL_MakeTempVector(int type, IDL_MEMINT dim, int init,
                         IDL_VPTR *var)where:
```

type, init, var

These arguments are the same as for **IDL_MakeTempArray()**.

dim

The number of elements in the resulting vector.

Creating a Temporary Structure

The **IDL_MakeTempStruct()** allows you to create an IDL structure variable using memory allocated by IDL, in much the same way that **IDL_MakeStruct()** and **IDL_ImportArray()** allow you to create an IDL structure variable using memory you provide. Temporary structure variables can be obtained via the **IDL_MakeTempStruct()** function:

```
char *IDL_MakeTempStruct(IDL_StructDefPtr sdef, int n_dim,
                        IDL_MEMINT dim[], IDL_VPTR *var, int zero)
```

where:

sdef

A pointer to the structure definition.

n_dim

The number of structure dimensions. The constant **IDL_MAX_ARRAY_DIM** defines the upper limit of this value.

dim

A C array of **IDL_MAX_ARRAY_DIM** elements containing the structure dimensions. The number of dimensions in the array is given by the **n_dim** argument.

var

The address of an **IDL_VPTR** where the address of the resulting temporary variable will be put.

The data area of an array **IDL_VARIABLE** is accessible from its **IDL_VPTR** as **var->value.arr->data**. However, since most routines that create an array need to access the data area, **IDL_MakeTempStruct()** returns the data area pointer as its value. As with **IDL_Gettmp()**, the variable allocated via **IDL_MakeTempStruct()** must be returned to the pool of temporary variables (with a call to **IDL_Deltmp()**) or be returned as the value of a system function before control returns to the interpreter, or an error will occur.

zero

Set to TRUE if the data area of the resulting variable should be zeroed, or to FALSE otherwise. Unless the caller intends to immediately copy a valid result into the variable, this argument should be set to TRUE to prevent memory corruption.

Creating a Temporary Vector

IDL_MakeTempStruct() can be used to create arrays with any number of dimensions, but the common case of creating a 1-dimensional vector can be carried out more conveniently using the **IDL_MakeTempStructVector()** function:

```
char *IDL_MakeTempStructVector(IDL_StructDefPtr sdef, IDL_MEMINT dim,
                               IDL_VPTR *var, int zero)
```

where:

sdef, var, zero

These arguments are the same as for **IDL_MakeTempStruct()**.

dim

The number of elements in the resulting vector.

Creating A Temporary Variable Using Another Variable As A Template

It is common to want to create a temporary variable with a form that mimics that of a variable you already have access to. Often, such a temporary variable has the same number of elements and dimensions, but may vary in type. It is possible to do this by using the basic temporary variable creation routines discussed earlier in this chapter, but the resulting code will be complex, and this sort of code occurs frequently. The best way to create such a variable is using the **IDL_VarMakeTempFromTemplate**() function.

IDL_VarMakeTempFromTemplate() creates a temporary variable of the desired type, using the **template_var** argument to specify its dimensionality. The address of this temporary variable is stored at the address specified by the **result_addr** argument. The address of the start of this variable's data area is returned as the value of the function.

```
char *IDL_VarMakeTempFromTemplate(IDL_VPTR template_var,int type,
                                  IDL_StructDefPtr sdef,
                                  IDL_VPTR *result_addr,int zero);
```

where:

template_var

Source variable to take dimensionality from. This can be a scalar or array of any type.

type

The IDL type code for the desired temporary variable.

sdef

NULL, or a pointer to a structure definition. This argument is ignored if **type** is not **IDL_TYP_STRUCT**. If type is **IDL_TYP_STRUCT**, **sdef** supplies the structure definition for the result. It is an error to specify a result type of **IDL_TYP_STRUCT** without providing a value for **sdef,** with one exception: If type is **IDL_TYP_STRUCT** and **template_var** is a variable of **IDL_TYP_STRUCT**, and **sdef** is NULL, then **IDL_VarMakeTempFromTemplate**() will use structure definition of **template_var.**

result_addr

Address of **IDL_VPTR** to receive a pointer to the newly allocated temporary variable.

zero

TRUE if the resulting variable should be zeroed, and FALSE to not do this. Variables of **IDL_TYP_STRING,** and structure types that contain strings, are always zeroed.

Freeing A Temporary Variable

Use **IDL_Deltmp()** to free a temporary variable:

```
void IDL_Deltmp(IDL_VPTR p)
```

where **p** is an **IDL_VPTR** to the temporary variable to be returned. **IDL_Deltmp()** frees the dynamic parts of the temporary variable (if any) and then returns the variable to the pool of available temporaries. Once you have deallocated a temporary variable, you may not access it again. There is also a macro named **IDL_DELTMP** which checks its argument to make sure it’s a temporary, and if so, calls **IDL_Deltmp()** to return it.

Creating an Array from Existing Data

There are two functions that allow you to create an IDL array variable whose data points at data you supply rather than having IDL allocate the data space. The routine **IDL_ImportArray()** returns a temporary variable, while **IDL_ImportNamedArray()** returns a named variable in the current execution scope, creating the new variable if necessary. Your data must already exist in memory. The data area, which can be quite large, is not copied. These functions simply create variable and array descriptors that point to the data you supply and return the pointer to the resulting variable. Their definitions are:

```
IDL_VPTR IDL_ImportArray(int n_dim, IDL_MEMINT dim[], int type,
        UCHAR *data, IDL_ARRAY_FREE_CB free_cb, void *s)

IDL_VPTR IDL_ImportNamedArray(char *name, int n_dim,
        IDL_MEMINT dim[], int type, UCHAR *data,
        IDL_ARRAY_FREE_CB free_cb, void *s)

typedef void (* IDL_ARRAY_FREE_CB) (UCHAR *);
```

where:

name

The name of the variable to be created or modified.

n_dim

The number of dimensions in the array.

dim

An array of **IDL_MAX_ARRAY_DIM** elements, containing the size of each dimension.

type

The IDL type code describing the data. See “Type Codes” on page 258.

data

A pointer to your array data. Your data will not be modified unless the user explicitly modifies elements of the array using subscripts.

The temporary variable returned by **IDL_ImportArray()** can be used immediately in an expression, in which case the descriptors are freed immediately. It can also be assigned to a longer-lived variable using **IDL_VarCopy()**.

Note

IDL frees only the memory that it allocates for the descriptors, *not* the memory that you supply containing your data. You can arrange to be notified when IDL is finished with your data by using the **free_cb** argument, described below.

free_cb

If non-NULL, **free_cb** is a pointer to a function that will be called when IDL frees the array. This feature gives the caller a sure way to know when IDL is no longer referencing data. Use the called function to perform any required cleanup such as freeing dynamic memory or releasing shared or mapped memory. The called function should have no return value and should accept as its argument a **(uchar *)**, which is a pointer to the memory to be freed.

s

If the type of the variable is **IDL_TYP_STRUCT**, **s** points to the opaque structure definition, as returned by **IDL_MakeStruct()**.

Getting Dynamic Memory

Many programs need to get dynamic memory for some temporary calculation. In the C language, the functions **malloc()** and **free()** are used for this purpose, while other languages have their own facilities. IDL provides its own memory allocation routines (see "Dynamic Memory" on page 396). Use of such facilities within the IDL interpreter and the system routines can lead to the loss of usable dynamic memory. The following code fragment demonstrates how this can happen.

For example, assume that there is a need for 100 IDL_LONG integers:

```
char *c;

c = (char *) IDL_MemAlloc((unsigned) (sizeof(IDL_LONG) * 100)
                         (char *) 0, IDL_MSG_RET);
.
.
.
if (some_error_condition) IDL_Message(…, IDL_MSG LONGJMP,…);
.
.
.
IDL_MemFree((void *) c, (char *) 0, IDL_MSG_RET);
```

In the normal case, the allocated memory is released exactly as it should be. However, if an error causes the **IDL_Message()** function to be called, execution will return directly to the interpreter and this code will never have a chance to clean up. The dynamic memory allocated will therefore leak, and although it will continue to occupy space in the IDL processes, will not be used again.

The IDL_GetScratch Function

To solve this problem, use a temporary variable to obtain dynamic memory. Then, if an error should cause execution to return to the interpreter, the interpreter will reclaim the temporary variable and no dynamic memory will be lost. This frequently-needed operation is provided by the **IDL_GetScratch()** function:

```
char *IDL_GetScratch(IDL_VPTR *p, IDL_MEMINT n_elts,
                     IDL_MEMINT elt_size)
```

where:

p

The address of an **IDL_VPTR** that should be set to the address of the temporary variable allocated.

n_elts

The number of elements for which memory should be allocated.

elt_size

The length of each element, in bytes.

Once the need for the temporary memory has passed, it should be returned using the **IDL_Deltmp()** function. Using these functions, the above example becomes:

```
char *c;
IDL_VPTR v;

c = IDL_GetScratch(&v, 100L, (IDL_LONG) sizeof(IDL_LONG));
.
.
.
if (some error condition) IDL_Message(...,MSG LONGJMP,...);
.
.
.
IDL_Deltmp(v);
```

Using the **IDL_GetScratch()** and **IDL_Deltmp()** functions is similar to using **IDLMemAlloc()** directly. In fact, IDL uses **IDL_MemAlloc()** and **IDL_MemFree()** internally to implement array variables. The important difference is that dynamic memory doesn’t leak when error conditions occur.

To avoid losing dynamic memory, always use the **IDL_GetScratch()** function in preference to other ways of allocating dynamic memory, and use **IDL_Deltmp()** to return it.

Accessing Variable Data

Often, we are not concerned with the distinction between a scalar and array variable—all that is desired is a pointer to the data and to know how many elements there are. **IDL_VarGetData()** can be used to obtain this information:

```
void IDL_VarGetData(IDL_VPTR v, IDL_MEMINT *n, char **pd,
                    int ensure_simple)
```

where:

v

The variable for which data is desired.

n

The address of a variable that will hold the number of elements.

pd

The address of variable that will hold a pointer to data, cast to be a pointer to a pointer to a character (for example (**char ****) **&myptr**).

ensure_simple

If TRUE, this routine calls the **IDL_ENSURE_SIMPLE** macro on the argument **v** to screen out variables of the types it prevents. Otherwise, **IDL_EXCLUDE_FILE** is called, because file variables have no data area to return.

On exit, **IDL_VarGetData()** stores the data count and pointer into the variables pointed at by **n** and **pd**, respectively.

Copying Variables

To copy the contents of one variable to another, use the **IDL_VarCopy()** function:

```
void IDL_VarCopy(IDL_VPTR src, IDL_VPTR dst)
```

Arguments src and dst are the source and destination, respectively.

IDL_VarCopy() uses the following rules when copying variables:

- If the destination variable already has a dynamic part, this dynamic part is released.
- The destination becomes a copy of the source.
- If the source is a temporary variable, **IDL_VarCopy()** does not make a duplicate of the dynamic part for the destination. Instead, the dynamic part of the source is given to the destination, and the source variable itself is returned to the pool of free temporary variables. This is the equivalent of freeing the temporary variable. Therefore, the variable must not be used any further and the caller should not explicitly free the variable. This optimization significantly improves resource utilization and performance because this special case occurs frequently.

Storing Scalar Values

The **IDL_StoreScalar()** function sets an **IDL_VARIABLE** to a scalar value:

```
void IDL_StoreScalar(IDL_VPTR dest, int type,
                     IDL_ALLTYPES *value)
```

where:

dest

An **IDL_VPTR** to the **IDL_VARIABLE** in which the scalar should be stored.

type

The type code for the scalar value. See “Type Codes” on page 258.

value

The address of the IDL_ALLTYPES union that contains the value to store.

If dest is a location that cannot be stored into (for example, a temporary variable, constant, and so on), an error is issued and control returns to the interpreter. Otherwise, any dynamic part of dest is freed and value is stored into it.

The **IDL_StoreScalarZero()** function is a specialized variation of **IDL_StoreScalar()**. It stores a zero scalar value of any specified type into the specified variable:

```
void IDL_StoreScalarZero(IDL_VPTR dest, int type)
```

where:

dest

An IDL_VPTR to the IDL_VARIABLE in which the scalar zero should be stored.

type

The type code for the scalar zero value. See “Type Codes” on page 258.

Using IDL_StoreScalar() to Free Dynamic Resources

In addition to performing its primary function, **IDL_StoreScalar()** and **IDL_StoreScalarZero()** have two very useful side effects:

1. Storing a scalar value in a variable causes IDL to free any dynamic memory currently used by that variable.
2. These routines do the required error checking to make sure the variable allows a new value to be stored into it before performing the actual storage operation.

Often, a system routine accepts an input argument that will have a new value assigned to it before the routine returns to its caller, and the initial value of that argument is of no interest to the routine. Storing a scalar value into such an argument at the start of the routine will automatically check it for storability and free unnecessary dynamic memory. In one easy operation, the required error checking is done, and you’ve improved the dynamic memory behavior of the IDL system by minimizing dynamic memory fragmentation. For example:

```
IDL_StoreScalarZero(&v, IDL_TYP_LONG);
```

Error handling is discussed further in “IDL Internals: Error Handling” on page 335.

Obtaining the Name of a Variable

The **IDL_VarName()** function returns the name of a variable, constant, or expression given its address. If the item is a named variable, it must be in the currently active program unit:

```
char *IDL_VarName(IDL_VPTR v)
```

Looking Up Main Program Variables

The **IDL_GetVarAddr()** function returns the address of a *main program* variable, given its name:

```
IDL_VPTR IDL_GetVarAddr(char *name)
```

name

Points to the null terminated name of the variable, which must be in upper case.

The return value is NULL if the variable does not exist, otherwise the pointer to the variable is returned.

Alternatively, **IDL_GetVarAddr1()** will enter a new variable into the symbol table of the main program if called with the parameter **ienter** set to TRUE, and the specified variable name does not already exist. Otherwise, its operation is the same as **IDL_GetVarAddr()**. Note that new variables cannot be created if a user procedure or function is active. **IDL_GetVarAddr1()** is called as shown following:

```
IDL_VPTR IDL_GetVarAddr1(char *name, int enter)
```

name

Points to the null-terminated name of the variable, which must be in upper case.

ienter

Set this parameter to TRUE to create the variable if it does not already exist.

If **ienter** is TRUE and the specified variable name does not already exist, it will be added to the symbol table of the main program. If **ienter** is FALSE, **IDL_GetVarAddr1()** is equivalent to **IDL_GetVarAddr()**.

Looking Up Variables in Current Scope

The **IDL_FindNamedVariable()** function returns the address of a variable in the *current execution scope* given its name:

```
IDL_VPTR IDL_FindNamedVariable(char *name, int ienter)
```

name

Name of the variable to find.

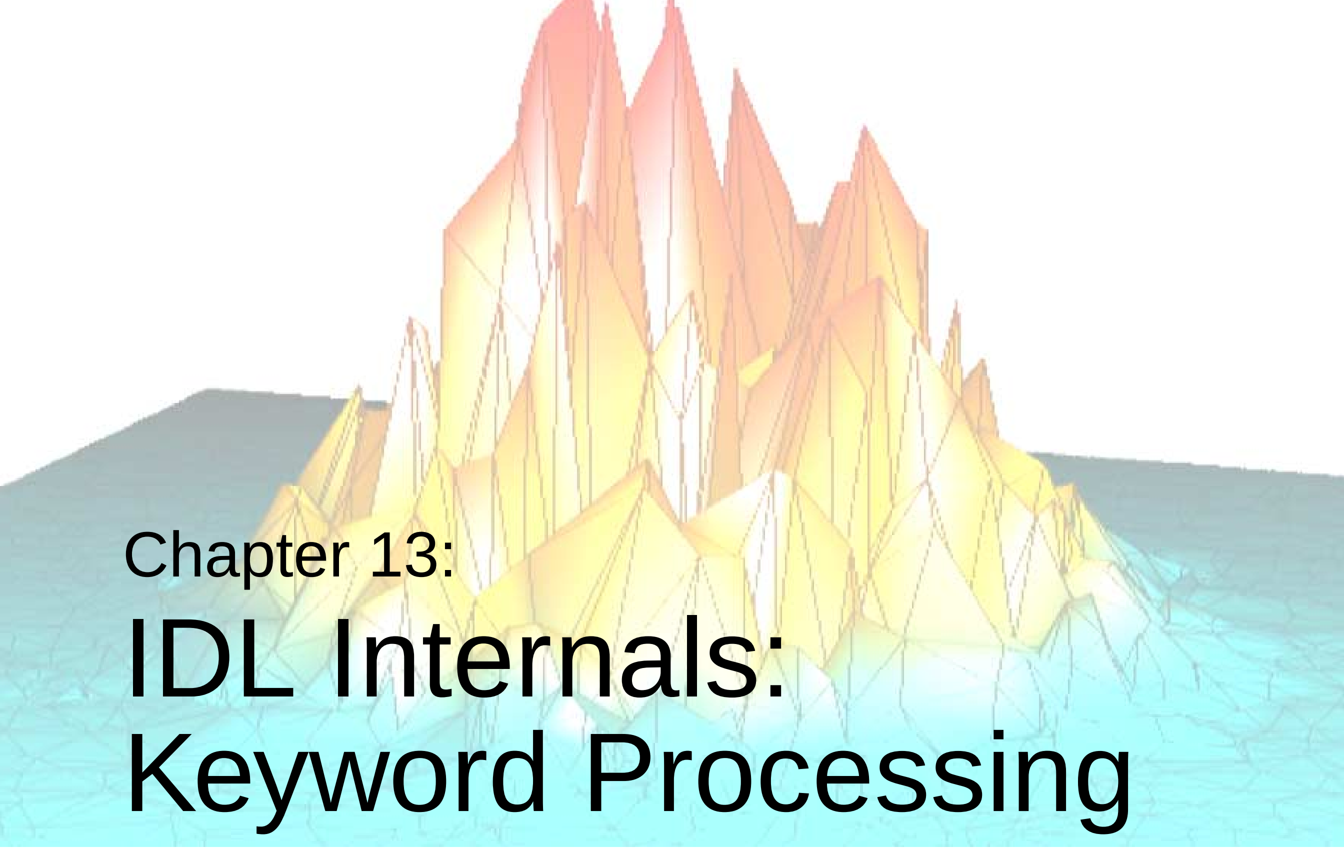
ienter

Set this parameter to TRUE to create the variable if it does not already exist.

If the variable is found (or created if **ienter** is TRUE), its **IDL_VPTR** is returned. Otherwise, NULL is returned.

Note

Even if **ienter** is TRUE, this routine can return NULL if creating the variable is not possible due to memory constraints.



Chapter 13: IDL Internals: Keyword Processing

This chapter discusses the following topics:

IDL and Keyword Processing

Keyword arguments are an important IDL language feature. They allow a multitude of options to be specified to a routine in a straightforward, easily understood way. The price of this added power is that it is somewhat more complicated to write a routine that accepts keywords than one that doesn’t. However, the additional effort is well worth it.

Creating Routines that Accept Keywords

As described in “Adding System Routines” on page 413, you must register your system routine before IDL will recognize it. When registering the routine, you indicate that it accepts keyword arguments in one of the following ways:

- Specifying the KEYWORDS option for the routine in the module definition file of a Dynamically Loadable Module (DLM)
- Setting the KEYWORDS keyword in a call to LINKIMAGE.
- OR-ing the constant **IDL_SYSFUN_DEF_F_KEYWORDS** into the **flags** field of the **IDL_SYSFUN_DEF2** struct passed to **IDL_SysRtnAdd()**

Routines that accept keywords must perform keyword processing. A routine that does not allow keyword processing knows that its **argc** argument gives the number of positional arguments, and **argv** contains only those positional arguments. In contrast, a routine that accepts keywords receives an **argc** that gives the total number of positional and keyword arguments, and these arguments are delivered in **argv** mixed together in an undefined order.

The function **IDL_KWProcessByOffset()** is used to process keywords and separate the positional and keyword arguments. It is passed an array of **IDL_KW_PAR** structures that give information about the allowed keywords and their attributes. The keyword data resulting from this process is stored in a user defined **KW_RESULT** structure. Finally, the **IDL_KW_FREE** macro is used to clean up.

More information about these routines and structures can be found in the following sections.

Overview Of IDL Keyword Processing

IDL keyword processing can seem confusing at first glance, due to the interrelated data structures involved. However, as the examples that follow in this chapter will show, the concepts involved are relatively straightforward once you have seen and understood a concrete example such as "Keyword Examples" on page 313.

Following is a skeleton of a system routine that accepts keyword arguments. These elements must be present in any such system routine:

```
void keyword_sysrtn_skeleton(int argc, IDL_VPTR *argv, char *argk)
{
  typedef struct {
    IDL_KW_RESULT_FIRST_FIELD; /* Must be first entry in struct */
    ...          /* Variables specific to your keywords go here */
  } KW_RESULT;
  static IDL_KW_PAR kw_pars[] = {
    /*
     * Keyword definitions for the keywords you accept go here,
     * one definition per keyword. The keyword definitions refer
     * to fields within the KW_RESULT type defined above.
     */
    ...
    { NULL }                    /* List must be NULL terminated */
  };
  KW_RESULT kw;  /* Variable which will hold the keyword values */

  (void) IDL_KWProcessByOffset(argc, argv, argk, kw_pars,
                               (IDL_VPTR *) 0, 1, &kw);

  /* The body of your routine */

  IDL_KW_FREE;
}
```

IDL keyword processing is made up of the following data structures and steps:

- A NULL terminated array of **IDL_KW_PAR** structures must be present. Each entry in this array describes the keyword processing required for a single keyword.
- If a keyword represents an input-only, by-value array, the **IDL_KW_PAR** structure that describes it points at an auxiliary **IDL_KW_ARR_DESC_R** structure that supplies the additional array specific information.
- The system routine must declare a local type definition named **KW_RESULT**, and a variable of this type named **kw**. The **KW_RESULT** type contains all of

the data fields that will be set as a result of processing the keywords described by the **IDL_KW_PAR** and **IDL_KW_ARR_DESC_R** structures described above. The **IDL_KW_PAR** and **IDL_KW_ARR_DESC_R** structures refer to the fields of the **KW_RESULT** structure by their offset from the beginning of the structure. The **IDL_KW_OFFSETOF**() macro is used to compute this offset.

- The system routine calls the **IDL_KWProcessByOffset**() function, passing it the address of the **IDL_KW_PAR** array, and the **KW_RESULT** variable (**kw**).
- After **IDL_KWProcessByOffset**() is called, the **KW_RESULT** structure (**kw**) contains the results, which can be accessed freely by the system routine.
- Before returning, the system routine must invoke the **IDL_KW_FREE** macro. This macro ensures that any dynamic memory used by **IDL_KWProcessByOffset**() is properly released.
- System routines are not required to, and generally do not, call **IDL_KW_FREE** before throwing errors using **IDL_Message**() with the **IDL_MSG_LONGJMP** or **IDL_MSG_IO_LONGJMP** action codes. In these cases, the IDL interpreter automatically knows to release the resources used by keyword processing on your behalf.

All of these data structures and routines are discussed in detail in the sections that follow.

The IDL_KW_PAR Structure

The **IDL_KW_PAR** struct provides the basic specification for keyword processing. The **IDL_KWProcessByOffset()** function is passed a null-terminated array of these structures. **IDL_KW_PAR** structures specify which keywords a routine accepts, the attributes required of them, and the kinds of processing that should be done to them. **IDL_KW_PAR** structures must be defined in lexical order according to the value of the **keyword** field.

The definition of **IDL_KW_PAR** is:

```
typedef struct {
  char *keyword;
  UCHAR type;
  unsigned short mask;
  unsigned short flags;
  int *specified;
  char *value;
} IDL_KW_PAR;
```

where:

keyword

A pointer to a null-terminated string. This is the name of the keyword, and must be entirely upper case. The array of **IDL_KW_PAR** structures passed to **IDL_KWProcessByOffset()** must be lexically sorted by the strings pointed to by this field. The final element in the array is signified by setting the keyword field to NULL (**(char *) 0**).

type

IDL_KWProcessByOffset() automatically converts the keywords to the IDL type specified by the **type** field. Specify 0 (**IDL_TYPE_UNDEF**) in cases where **ID_KW_VIN** or **IDL_KW_OUT** are specified in the **flags** field.

mask

The enable mask. This field is ANDed with the mask argument to **IDL_KWProcessByOffset()** and if the result is non-zero, the keyword is accepted. If the result is 0, the keyword is ignored. This ability allows you to share an array of **IDL_KW_PAR** structures between several routines, and enable or disable the keywords used by each one.

As an example of this, the IDL graphics and plotting routines have a large number of keywords in common. In addition, each routine has a few keywords that are unique to it. Keywords are implemented using a single shared array of **IDL_KW_PAR** with appropriate values of the mask field. This technique dramatically reduces the amount of data that would otherwise be required by graphics keyword processing, and makes IDL easier to maintain.

flags

This field specifies special processing instructions. It is a bit mask made by ORing the following values:

- **IDL_KW_ARRAY** — Set this bit to specify that the keyword must be an array. Otherwise, a scalar is required. If **IDL_KW_ARRAY** is specified, the value field must point at an associated **IDL_KW_ARR_DESC_R** structure.
- **IDL_KW_OUT** — Set this bit to indicate that the keyword specifies an output parameter, passed by reference. Expressions and constants are excluded. In other words, the routine is going to change the value of the keyword argument, as opposed to the more usual case of simply reading it. The address of the **IDL_VARIABLE** will be placed in a user supplied field of type **IDL_VPTR** in the **KW_RESULT** structure (**kw**). The offset of this field in the **KW_RESULT** structure is specified by the **value** field (discussed below). **IDL_KW_OUT** implies that no type checking or processing will be performed on the keyword—it is up to the routine to perform the same sort of type checking normally carried out for plain positional arguments.

  A standard approach to find out if an **IDL_KW_OUT** parameter is present in a call to a system routine is to specify **IDL_TYP_UNDEF** (0) for the type field and **IDL_KW_OUT | IDL_KW_ZERO** for flags. The **IDL_VPTR** referenced by the value field will either contain NULL, or a pointer to the **IDL_VARIABLE**.
- **IDL_KW_VIN** — Set this bit to indicate that the keyword parameter is an input parameter (expressions and/or constants are valid) passed by reference. The address of the **IDL_VARIABLE** or expression is stored in a user-supplied field of the **KW_RESULT** structure (**kw**) referenced by the value field, as with **IDL_KW_OUT**. **IDL_KW_VIN** implies that no type checking or processing will be performed on the keyword—it is up to the routine to perform the same sort of type checking normally carried out for plain positional arguments.
- **IDL_KW_ZERO** — Set this bit in order to *zero* the C variable pointed to by the value field before parsing the keywords. This means that the object pointed

to by value will always be zero unless it was specified by the user. Use this technique to create keywords that have Boolean (on or off) meanings.

- **IDL_KW_VALUE** — If this bit is set and the specified keyword is present and non-zero, the low 12 bits of this field (**flags**) will be bitwise ORed with the **IDL_LONG** field of the KW_RESULT structure referenced by the **value** field. Note that this implies the **IDL_TYP_LONG** type code, and is incompatible with the **IDL_KW_ARRAY, IDL_KW_VIN,** and **IDL_KW_OUT** flags.

specified

NULL, or the offset of the user supplied field within the **KW_RESULT** structure (**kw**) of a C int variable that will be set to TRUE (non-zero) or FALSE (0) based on whether the routine was called with the keyword present. The **IDL_KW_OFFSETOF**() macro should be used to calculate the offset. Setting this field to NULL (0) indicates that this information is not needed.

value

If the keyword is a read-only scalar, this field is the offset of the user supplied field in the **KW_RESULT** structure (**kw**) into which the keyword value will be copied. The **IDL_KW_OFFSETOF**() macro should be used to calculate the offset.

In the case of a read-only array, value is the memory address of an **IDL_KW_ARR_DESC_R**, structure, which is discussed in "The IDL_KW_ARR_DESC_R Structure" on page 305.

In the case of an input (**IDL_KW_VIN**) or output (**IDL_KW_OUT**) variable, this field should contain the offset to the **IDL_VPTR** field within the user supplied **KW_RESULT** that will be filled by **IDL_KWProcessByOffset()** with the address of the keyword argument. The **IDL_KW_OFFSETOF()** macro should be used to calculate the offset.

The IDL_KW_ARR_DESC_R Structure

When a keyword is specified to be a read-only array (i.e., the **IDL_KW_ARRAY** flag is set), the value field of the **IDL_KW_PAR** struct should be set to point to an **IDL_KW_ARR_DESC_R** structure. This structure is defined as:

```
typedef struct {
  char *data;
  IDL_MEMINT nmin;
  IDL_MEMINT nmax;
  IDL_MEMINT n_offset;
} IDL_KW_ARR_DESC_R;
```

where:

data

The offset of the field within the user supplied **KW_RESULT** structure, of the C array to receive the data. This offset is computed using the **IDL_KW_OFSETOF()** macro. This array must be of the C type specified by the **type** field of the **IDL_KW_PAR** struct. For example, **IDL_TYP_LONG** maps into a C **IDL_LONG**. There must be **nmax** elements in the array.

nmin

The minimum number of elements allowed.

nmax

The maximum number of elements allowed.

n_offset

The offset of the field within the user defined **KW_RESULT** structure into which **IDL_KWProcessByOffset**() will store the number of elements actually stored into the array field. This offset is computed using the **IDL_KW_OFSETOF()** macro.

Keyword Processing Options

The following cases occur in keyword processing:

Scalar Input-Only

For scalar, input-only keywords, the user never sees the **IDL_VARIABLE** passed as the keyword argument. Instead, the value of the **IDL_VARIABLE** is converted to the type specified by the **type** field of the **IDL_KW_PAR** struct and is placed into the field of the user specified **KW_RESULT** structure, the offset of which is given by the **value** field. This offset is calculated using the **IDL_KW_OFFSETOF**() macro.

Array Input-Only

Array input-only keywords work similarly to the scalar case, except that the **value** field contains the address of an **IDL_KW_ARR_DESC_R** struct that supplies the added information required to convert the passed array elements to the specified type and place them into a C array for easy access. The array data is copied into a array within the user supplied **KW_RESULT** structure. The **data** field of the **IDL_KW_ARR_DESC_R** struct supplies the offset of the array field within the **KW_RESULT** structure. This offset is calculated using the **IDL_KW_OFFSETOF**() macro.

As part of this process, the number of array elements passed is checked to be within the range specified in the **IDL_KW_ARR_DESC_R** struct, and if no error results, the number is stored into a field of the user supplied **KW_RESULT** struct. The **n_offset** field of the **IDL_KW_ARR_DESC_R** struct supplies the offset of this "number of elements" field within the **KW_RESULT** structure. This offset is calculated using the **IDL_KW_OFFSETOF**() macro.

It is worth noting that input-only array keywords don't pass information about the number of dimensions or their sizes, only the total number of elements. Therefore, they are treated as 1-dimensional vectors. For more flexibility, use an Input/Output keyword instead.

Input/Output

This case occurs if the **IDL_KW_VIN** or **IDL_KW_OUT** flag is set in the **IDL_KW_PAR** struct. In this case, the value field contains the offset of the **IDL_VPTR** field (computed with the **IDL_KW_OFFSETOF()** macro) in the user defined **KW_RESULT** struct into which the actual keyword argument is copied. In this case, you must do all error checking and type conversion yourself, just like with positional arguments. This is certainly the most flexible method. However, the other two cases are much easier to use, and are suitable for the vast majority of keywords.

The KW_RESULT Structure

Each system routine that processes keywords is required to define a structure variable into which **IDL_KWProcessByOffset**() will store all the results of keyword processing. This variable must follow the following rules:

- The name of the structure type must be defined as **KW_RESULT**. This requirement exists so that the **IDL_KW_OFFSETOF**() macro can properly do its work.
- The first field within any **KW_RESULT** structure must be defined using the **IDL_KW_RESULT_FIRST_FIELD** macro. The contents of this first field are private, and should not be examined. It contains the information required by IDL to properly track its resource use.
- The name of the **KW_RESULT** variable must be **kw**. This requirement exists so that the **IDL_KW_FREE** macro can properly do its work.

Hence, all system routines that process keywords will contain statements similar to the following:

```
typedef struct {
    IDL_KW_RESULT_FIRST_FIELD;/* Must be first entry in struct */
    …                       /* Additional user specified fields */
  } KW_RESULT;

  KW_RESULT kw;
```

All fields within the **KW_RESULT** structure after the required first field can have arbitrary user selected names. The types of these fields are dictated by the **IDL_KW_PAR** and **IDL_KW_ARR_DESC_R** structures that refer to them.

Processing Keywords

The **IDL_KWProcessByOffset()** function is used to process keywords. **IDL_KWProcessByOffset()** performs the following actions on behalf of the calling system routine:

- Verify that the keywords passed to the routine are all allowed by the routine.
- Carry out the type checking and conversions required for each keyword.
- Find the positional (non-keyword) arguments that are scattered among the keyword arguments in **argv** and copy them in order into the **plain_args** array.
- Return the number of plain arguments copied into **plain_args**.

IDL_KWProcessByOffset() has the form:

```
int IDL_KWProcessByOffset(int argc, IDL_VPTR *argv, char *argk,
                          IDL_KW_PAR *kw_list,
                          IDL_VPTR plain_args[], int mask,
                          void * base)
```

where:

argc

The number of arguments passed to the caller. This is the first parameter to all system routines.

argv

The array of **IDL_VPTR** to arguments that was passed to the caller. This is the second parameter to all system routines.

argk

The pointer to the keyword list that was passed to the caller. This is the third parameter to all system routines that accept keyword arguments.

kw_list

An array of **IDL_KW_PAR** structures (see "Overview Of IDL Keyword Processing" on page 300) that specifies the acceptable keywords for this routine. This array is terminated by setting the keyword field of the final struct to NULL (**(char *) 0**).

plain_args

NULL, or an array of **IDL_VPTR** into which the **IDL_VPTR**s of the positional arguments will be copied. This array must have enough elements to hold the maximum possible number of positional arguments, as defined in **IDL_SYSFUN_DEF2**. See "Registering Routines" on page 438.

Note

IDL_KWProcessByOffset() sorts the plain arguments into the front of the input **argv** argument. Hence, **plain_args** is often not necessary, and can be set to NULL.

mask

Mask enable. This variable is ANDed with the mask field of each **IDL_KW_PAR** struct in the array given by **kw_list**. If the result is non-zero, the keyword is accepted as a valid keyword for the called system routine. If the result is zero, the keyword is ignored.

base

Address of the user supplied **KW_RESULT** structure, which must be named **kw**.

Speeding Keyword Processing

As mentioned above, the **kw_list** argument to **IDL_KWProcessByOffset()** is a null terminated list of **IDL_KW_PAR** structures. The time required to scan each item of the keyword array and zero the required fields (those fields specified, and value fields with **IDL_KW_ZERO** set), can become significant, especially when more than a few keyword array elements (e.g., 5 to 10 elements) are present.

To speed things up, specify **IDL_KW_FAST_SCAN** as the first keyword array element. If **IDL_KW_FAST_SCAN** is the first keyword array element, the keyword array is compiled by **IDL_KWProcessByOffset()** into a more efficient form the first time it is used. Subsequent calls use this efficient version, greatly speeding keyword processing. Usage of **IDL_KW_FAST_SCAN** is optional, and is not worthwhile for small lists. For longer lists, however, the improvement in speed is noticeable. For example, the following list does not use fast scanning:

```
static IDL_KW_PAR  kw_pars[] = {
  { "DOUBLE", IDL_TYP_DOUBLE, 1, 0,
     IDL_KW_OFFSETOF(d_there), IDL_KW_OFFSET_OF(d) },
  { "FLOAT", IDL_TYP_FLOAT, 1,IDL_KW_ZERO,0,IDL_KW_OFFSET_OF(f) },
  { NULL }
};
```

To use fast scanning, it would be written as:

```
static IDL_KW_PAR  kw_pars[] = {
  IDL_KW_FAST_SCAN,
  { "DOUBLE", IDL_TYP_DOUBLE, 1, 0,
    IDL_KW_OFFSET_OF(d_there), IDL_KW_OFFSETOF(d) },
  {"FLOAT", IDL_TYP_FLOAT, 1, IDL_KW_ZERO, 0,IDL_KW_OFFSETOF(f) },
  { NULL }
};
```

Cleaning Up

All normal exit paths from your system routine are required to call the **IDL_KW_FREE** macro prior to returning. This macro must be called exactly once for every call to **IDL_KWProcessByOffset**(). You must therefore structure your code so that **IDL_KW_FREE** executes before any return statement. Many functions to not use an explicit return statement, relying on the implicit return that occurs when execution comes to the end of the function. In such a case, **IDL_KW_FREE** must be the last statement in the function.

Keyword Examples

The following C function implements KEYWORD_DEMO, a system procedure intended to demonstrate how to write the keyword processing code for a routine. It prints the values of its keywords, changes the value of READWRITE to 42 if it is present, and returns. Each line is numbered to make discussion easier. These numbers are not part of the actual program.

Note

The following code is designed to demonstrate keyword processing in a simple, uncluttered example. In actual code, you would not use the **printf** mechanism used on lines 42-53.

C

```
 1 void keyword_demo(int argc, IDL_VPTR *argv, char *argk)
 2 {
 3   typedef struct {
 4     IDL_KW_RESULT_FIRST_FIELD; /* Must be first entry in structure */
 5     IDL_LONG l;
 6     float f;
 7     double d;
 8     int d_there;
 9     IDL_STRING s;
10     int s_there;
11     IDL_LONG arr_data[10];
12     int arr_there;
13     IDL_MEMINT arr_n;
14     IDL_VPTR var;
15   } KW_RESULT;
16   static IDL_KW_ARR_DESC_R arr_d = { IDL_KW_OFFSETOF(arr_data), 3, 10,
17                                      IDL_KW_OFFSETOF(arr_n) };
18 
19   static IDL_KW_PAR kw_pars[] = {
20     IDL_KW_FAST_SCAN,
21     { "ARRAY", IDL_TYP_LONG, 1, IDL_KW_ARRAY,
22       IDL_KW_OFFSETOF(arr_there), CHARA(arr_d) },
23     { "DOUBLE", IDL_TYP_DOUBLE, 1, 0,
24       IDL_KW_OFFSETOF(d_there), IDL_KW_OFFSETOF(d) },
25     { "FLOAT", IDL_TYP_FLOAT, 1, IDL_KW_ZERO, 0, IDL_KW_OFFSETOF(f) },
26     { "LONG", IDL_TYP_LONG, 1, IDL_KW_ZERO|IDL_KW_VALUE|15, 0,
27       IDL_KW_OFFSETOF(l) },
28     { "READWRITE", IDL_TYP_UNDEF, 1, IDL_KW_OUT|IDL_KW_ZERO,
29       0, IDL_KW_OFFSETOF(var) },
30     { "STRING", TYP_STRING, 1, 0,
31       IDL_KW_OFFSETOF(s_there), IDL_KW_OFFSETOF(s) },
32     { NULL }
33   };
34 
```

Figure 13-1: Keyword processing example.

C
```
  KW_RESULT kw;
  int i;
  IDL_ALLTYPES newval;

  (void) IDL_KWProcessByOffset(argc, argv, argk, kw_pars,
                               (IDL_VPTR *) 0, 1, &kw);

  printf("LONG: <%spresent>\n", kw.l ? "": "not ");
  printf("FLOAT: %f\n", kw.f);
  printf("DOUBLE: <%spresent>\n", kw.d_there ? "": "not ");
  printf("STRING: %s\n",
         kw.s_there ? IDL_STRING_STR(&kw.s) : "<not present>");
  printf("ARRAY: ");
  if (kw.arr_there)
  for (i = 0; i < kw.arr_n; i++)
  printf(" %d", kw.arr_data[i]);
  else
  printf("<not present>");
  printf("\n");

  printf("READWRITE: ");
  if (kw.var) {
  IDL_Print(1, &kw.var, (char *) 0);
  newval.l = 42;
  IDL_StoreScalar(kw.var, TYP_LONG, &newval);
  } else {
  printf("<not present>");
  }
  printf("\n");

  IDL_KW_FREE;
}
```

Figure 13-1: (Continued) Keyword processing example.

Executing this routine from the IDL command line, by entering:

```
KEYWORD_DEMO
```

gives the output:

```
LONG: <not present>
FLOAT: 0.000000
DOUBLE: <not present>
STRING: <not present>
ARRAY: <not present>
READWRITE: <not present>
```

Executing it again with keywords specified:

```
A = 56
KEYWORD_DEMO, /LONG, FLOAT=2, DOUBLE=34,$
  STRING="hello", ARRAY=FINDGEN(10), READWRITE=A
PRINT, 'Final Value of A: ', A
```

gives the output:

```
LONG: <present>
FLOAT: 2.000000
DOUBLE: <present>
STRING: hello
ARRAY: 0 1 2 3 4 5 6 7 8 9
READWRITE:      56
Final Value of A:           42
```

Those features of this procedure that are interesting in terms of keyword processing are, by line number:

3-15

Every system routine that processes keywords must define a **KW_RESULT** structure type. All output from keyword processing is stored in the fields of this structure.The first field in the **KW_RESULT** structure must always be **IDL_KW_RESULT_FIRST_FIELD**. The remaining fields are dictated by the keywords defined in **kw_pars** below, starting on line 19. The fields with named ending in **_there** are used for the specified field of the **IDL_KW_PAR** structs, and must be type int. The types of the other fields must match their definitions in the relevant **IDL_KW_PAR** and **IDL_KW_ARR_DESC_R** structs.

16-17

The ARRAY keyword, defined on line 21, is a read-only array, and requires this array description. Note that the data field specifies the location of the **arr_data** array within **KW_RESULT** where the array contents should be copied, and the **n_offset** field specifies the location of the **arr_n** field where the number of elements actually seen is to be written. Both of these are specified as offsets into **KW_RESULT**, using the **IDL_KW_OFFSET**() macro to compute this. The minimum number of elements allowed is 3, the maximum is 10.

19

The start of the keyword definition array. Notice that all of the keywords are ordered lexically (ASCII) and that there is a NULL entry at the end (line 32). Also, all of the mask fields are set to 1, as is the mask argument to **IDL_KWProcessByOffset**() on line 39. This means that all of the keywords in the list are to be considered valid in this routine.

20

This routine is requesting fast keyword processing. You can learn more about this option in "Speeding Keyword Processing" on page 310.

21-22

ARRAY is a read-only array. Its value field is therefore the actual address (and not an offset into **KW_RESULT**) of the **IDL_KW_ARR_DESC_R** struct that completes the array definition. This program wants to know explicitly if ARRAY was specified, so the **specified** field is set to the offset within **KW_RESULT** of the **arr_there** field.

23-24

DOUBLE is a scalar keyword of **IDL_TYP_DOUBLE**. It uses the variable **kw.d_there** to know if the keyword is present. Both the specified and value fields are specified as offsets into **KW_RESULT**.

25

FLOAT is an **IDL_TYP_FLOAT** scalar keyword. It does not use the **specified** field of the **IDL_KW_PAR** struct to get notification of whether the keyword is present, so that field is set to 0. Instead, it uses the **IDL_KW_ZERO** flag to make sure that the variable **kw.f** is always zeroed. If the keyword is present, the value will be written into **kw.f**, otherwise it will remain 0. The important point is that the routine can't tell the difference between the keyword being absent, or being present with a user-supplied value of zero. If this distinction doesn't matter, such as when the keyword is to serve as an on/off toggle, use this method. If it does matter, use the **specified** field as demonstrated with the DOUBLE keyword, above.

26-27

LONG is a scalar keyword of **IDL_TYP_LONG**. It sets the **IDL_KW_ZERO** flag to get the variable **kw.l** zeroed prior to keyword parsing. The use of the **IDL_KW_VALUE** flag indicates that if the keyword is present, the value 15 (the lower 12 bits of the flags field) will be ORed into the variable **kw.l**.

28-29

The **IDL_KW_OUT** flag indicates that the routine wants the **IDL_VPTR** for READWRITE if it is present. Since **IDL_KW_ZERO** is also set, the variable

kw.var will be zero unless the keyword is present. The specification of **IDL_TYP_UNDEF** here indicates that there is no type conversion or processing applied to **IDL_KW_OUT** keywords.

30-31

The STRING keyword demonstrates scalar string keywords.

32

All **IDL_KW_PAR** arrays must be terminated with a NULL entry.

35

Every system routine that processes keywords must declare a variable named **kw**, of type **KW_RESULT**. This variable should be a C stack based local variable (C auto class).

37

The **IDL_StoreScalar()** function used on line 59 requires the scalar value to be provided in an **IDL_ALLTYPES** struct.

39-40

Do the keyword processing. The first three arguments are simply the arguments the interpreter passed to the routine. The **plain_args** argument is set to NULL because this routine is registered as not accepting any plain arguments. Since no plain arguments will be present, the return value from **IDL_KWProcessByOffset()** is discarded. The final argument is the address of the **KW_RESULT** variable (**kw**) into which the results will be written.

42

The **kw.l** variable will be 0 if LONG is not present, and 1 if it is.

43

The **kw.f** variable will always have some usable value, but if it is zero there is no way to know if the keyword was actually specified or not.

44-46

These keywords use the variables from the **specified** field of their **IDL_KW_PAR** struct to determine if they were specified or not. Use of the **IDL_STRING_STR** macro is described in “Accessing IDL_STRING Values” on page 329.

47-53

Accessing the ARRAY keyword is simple. The **kw.arr_there** variable indicates if the keyword is present, and **kw.arr_n** gives the number of elements.

55-63

Since the READWRITE keyword is accessed via the argument’s **IDL_VPTR**, we use the **IDL_Print()** function to print its value. This has the same effect as using the user-level PRINT procedure when running IDL. See “Output of IDL Variables” on page 384. Then, we change its value to 42 using **IDL_StoreScalar()**.

Again, please note that we use this mechanism in order to create a simple example. You will probably want to avoid the use of this type of output (**printf** and **IDL_Print()**) in your own code.

65

Normal exit from any routine that calls **IDL_KWProcessByOffset()** must be preceded by a call to **IDL_KW_FREE**. This macro releases any dynamic resources that were allocated by keyword processing.

The Pre-IDL 5.5 Keyword API

Versions of IDL prior to IDL 5.5 used a different, but similar, keyword processing API to that found in the current versions. The remainder of this chapter provides information of interest to programmers maintaining older system routines that were written to that API.

Note

Research System recommends that all new code be written using the new keyword processing API. The older API continues to be supported for backwards compatibility, and there is no urgent reason to convert code that uses it. However, the effort of converting old code to the new API is minimal, and can be beneficial.

Background

If you have system routines that were written for use with versions of IDL older than IDL 5.5, your code uses an older keyword processing API, described in "Processing Keywords With IDL_KWGetParams()" on page 520, that including the following obsolete elements:

- **IDL_KWGetParams()**
- **IDL_KW_ARR_DESC**
- **IDL_KWCleanup()**, **IDL_KW_MARK**, **IDL_KW_CLEAN**

This old API served for many years, but it had some unfortunate features that made it hard to use correctly:

- The rules for when and how to use **IDL_KWCleanup()** were difficult to remember. The programmer had to decide whether or not to call it based on the types of the keywords being processed. If you didn't call it when you should, memory would be leaked.
- In order to ensure correctness, many programmers would resort to always calling **IDL_KWCleanup()** whether it was is needed or not. This is inefficient, but otherwise fine.
- The use of **IDL_KWCleanup()** is based on a worst case assumption that the keywords that require cleaning will actually be invoked by the IDL user. This is often not the case, and is therefore inefficient.
- Imagine an existing system routine that does not need to use **IDL_KWCleanup()**, and therefore is coded not to use it. If a new keyword

should later be added to that routine, and that new keyword should require the use of **IDL_KWCleanup()**, it is very likely that the programmer adding this new keyword will fail to recognize that issue. This leads to memory leaking from a formerly correct routine.

- If a future version of IDL should get a new data type that requires cleaning, that will change the rules for when **IDL_KWCleanup()** needs to be called. Existing code may need to be examined to fix this situation.
- The old keyword API is not reentrant, because it requires static variable addresses to be embedded in the keyword list. This has always been a problem for recursively callable routines (*e.g.* WIDGET_CONTROL, which can cause an IDL procedure callback to execute, which can in turn call WIDGET_CONTROL again). In the past, we have carefully coded these complex routines to avoid problems, but the large amount of code required is difficult to write and verify. The proper solution is a reentrant keyword API that uses offsets to variables within a structure, instead of actual statically scoped variable addresses. This is what the current API provides.

Advantages Of The IDL 5.5 API

In contrast, keyword processing, in IDL 5.5 and later is built around the **IDL_KWProcessByOffset**() function, has the following advantages:

- The old API remains in place with full functionality. Hence, you are not required to update your old code (There are benefits to such updating, however).
- A transitional API, build around the **IDL_KWProcessByAddr()** function, exists to help ease updating your code. See "The Transitional API" on page 323 for details. The transitional API is a halfway measure designed to solve the worst problems of the old API while requiring the minimum amount of change.
- The new API, and the transitional API both eliminate the confusing **IDL_KWCleanup()** routine and its requirement to **KW_MARK** before**,** and **KW_CLEAN** after keyword processing based on the types of the keywords. Instead, the keyword processing API keeps track of the need to cleanup itself, and handles this efficiently. The user is freed from guesswork about how and when to do the required cleanup.
- Keyword cleanup will only happen if the keyword module determines that it is necessary as it processes the actual keywords used. This is more efficient, and

it can be easily extended within IDL if a new data type is added to the IDL system, without requiring any change to your code.

- The internal data structures used to maintaining keyword state are now dynamically allocated, and do not have the static limits that the old implementation did.
- The new API is able to process keywords using a re-entrant keyword description. Results are written to stack based (C auto) variables rather than statically defined variables. This can be used to greatly simplify the implementation of recursive system routines, and has been used extensively for that purpose within IDL.

Differences And Similarities Between APIs

The current IDL keyword processing API was designed to minimize the changes necessary to convert existing older code. The differences and similarities between these APIs are summarized below:

- The basic **IDL_KW_PAR** data structure is unchanged between the two. However, in the old API, the **specified**, and **value** fields are addresses to statically allocated C variables or **IDL_KW_ARR_DESC** structures. In the new API, **specified** is always an offset into a user defined **KW_RESULT** structure. The **value** field is an offset into **KW_RESULT** when it is used to refer to data. It is an address when used to refer to an **IDL_KW_ARR_DESC_R** structure.
- The old API uses the **IDL_KW_ARR_DESC** structure to define **IDL_KW_ARRAY** read-only arrays. The new API uses the very similar **IDL_KW_ARR_DESC_R** structure. This is necessary because **IDL_KW_ARR_DESC** is not reentrant (the number of array elements used is written into the struct), while **IDL_KW_ARR_DESC_R** causes them to be written into a field in the **KW_RESULT** variable instead.
- The new API requires all keyword variables to be contained in a single **KW_RESULT** structure, while the old API allowed them to be independent variables. This is important to the offset-based scheme used in the new API, as well as having the nice side effect of improving the organization and readability of most code.
- The old API uses **IDL_KWGetParams**() to process keywords. The new API uses **IDL_KWProcessByOffset()**.

- The old API uses **IDL_KWCleanup()** to free resources. The rules for using it are complicated and lead to latent coding errors. The new API uses the **IDL_KW_FREE** macro, and has a simple consistent rule for use.

Converting Existing Code To The New API

To convert code that uses the old API to the new version:

- Define a typedef for a struct named **KW_RESULT**, containing the keyword variables. Make the first field be the predefined **IDL_KW_RESULT_FIRST_FIELD**.
- Modify your keyword definition list so that the specified and value fields of each **IDL_KW_PAR** struct contain offsets instead of addresses in all cases except when the value field references an **IDL_KW_ARR_DESC** struct. To do this, use the **IDL_KW_OFFSETOF()** macro.
- Any reference to an **IDL_KW_ARR_DESC** structure for an **IDL_KW_ARRAY** keyword must be converted to an **IDL_KW_ARR_DESC_R** struct.
- Replace the call to **IDL_KWGetParams()** with a call to **IDL_KWProcessByOffset()**.
- Remove any I**DL_KWCleanup(IDL_KW_MARK)** calls.
- Replace any **IDL_KWCleanup(IDL_KW_CLEAN)** calls with the **IDL_KW_FREE** macro. Check to ensure that all exit paths from your function other than via **IDL_Message()** include a call to this macro.

The Transitional API

RSI recommends that your convert your code to the reentrant keyword API based around the I**DL_KWProcessByOffset()** and I**DL_KWFree()** functions. This is almost always a straightforward operation, and the resulting code has all of the advantages discussed in "Advantages Of The IDL 5.5 API" on page 321. However, there is another alternative that may be useful is some situations. A third keyword API, built around the **IDL_KWProcessByAddr()** function exists that provides the benefits of eliminating the confusing **IDL_KWCleanup()** function, while not requiring the use of static non-reentrant separate variables to change. The transitional API is a halfway measure designed to solve the worst problems of the old API while requiring the minimum amount of change to your code:

```
int IDL_KWProcessByAddr(int argc, IDL_VPTR *argv, char *argk,
                        IDL_KW_PAR *kw_list, IDL_VPTR *plain_args,
                                int mask, int *free_required)

void IDL_KWFree(void)
```

where:

argc, argv, argk, plain_args, mask

These arguments are the same as those required by **IDL_KWProcessByOffset()**

kw_list

An array of **IDL_KW_PAR** structures, in the absolute address form required by the old **IDL_KWGetParams()** keyword API (the **specified** and **value** fields use address to static C variables).

free_required

The address of an integer to be filled in by **IDL_KWProcessByAddr()**. If set to TRUE, the caller must call **IDL_KWFree()** prior to exit from the routine.

Example: Converting From The Old Keyword API

To illustrate the use of the old keyword API, the transitional API, and the new reentrant API, this section provides an extremely simple example, written three times, once with each API.

Another useful comparison is to compare the example "Keyword Examples" on page 313 with its original version written with the old API which can be found in "Keyword Examples" on page 524.

Old API

```
IDL_VPTR IDL_someroutine(int argc, IDL_VPTR *argv, char *argk)
{
  static IDL_VPTR count_var;
  static IDL_LONG debug;
  static IDL_STRING name;
  static IDL_KW_PAR kw_pars[] = {
    { "COUNT", 0,1,IDL_KW_OUT|IDL_KW_ZERO,0,IDL_CHARA(count_var)},
    { "DEBUG", IDL_TYP_LONG, 1, IDL_KW_ZERO, 0,IDL_CHARA(debug) },
    { "NAME", IDL_TYP_STRING, 1, IDL_KW_ZERO, 0,IDL_CHARA(name) },
    { NULL }
  };
  IDL_VPTR result;

  IDL_KWCleanup(IDL_KW_MARK);
  argc = IDL_KWGetParams(argc,argv,argk,kw_pars,(IDL_VPTR *)0,1);

  /* Your code goes here. Keyword values are available in the
   * static variables.*/

  /* Cleanup keywords before leaving */
  IDL_KWCleanup(IDL_KW_CLEAN);
  return(result);
}
```

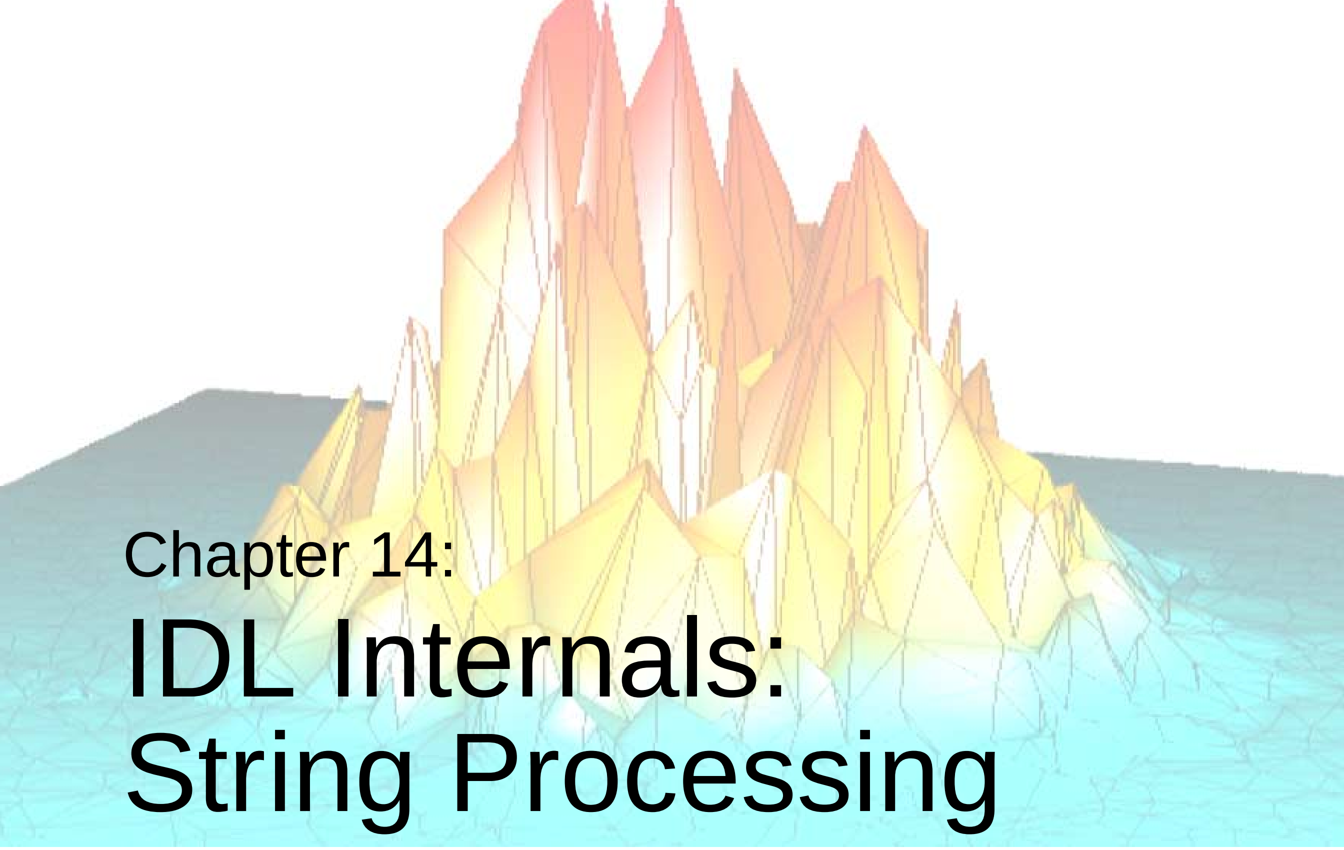
Transitional API

The transitional API provides the benefits of simplified and straightforward cleanup, but does not require you to alter your IDL_KW_PAR array or gather the keyword variables into a common structure. The resulting code is very similar to the old API.

```
IDL_VPTR IDL_someroutine(int argc, IDL_VPTR *argv, char *argk)
{
  static IDL_VPTR count_var;
  static IDL_LONG debug;
  static IDL_STRING name;
  static IDL_KW_PAR kw_pars[] = {
    {"COUNT", 0, 1, IDL_KW_OUT|IDL_KW_ZERO,
     0,IDL_KW_ADDROF(count_var) },
    { "DEBUG", IDL_TYP_LONG,1,IDL_KW_ZERO,0,IDL_KW_ADDROF(debug)},
    { "NAME", IDL_TYP_STRING,1,IDL_KW_ZERO,0,IDL_KW_ADDROF(name)},
    { NULL }
  };

  int kw_free;
  IDL_VPTR result;

  argc = IDL_KWProcessByAddr(argc, argv, argk, kw_pars,
                             (IDL_VPTR *) 0, 1, &kw_free);

  /* Your code goes here. Keyword values are available in the
   * static variables.*/

  /* Cleanup keywords before leaving */
  if (kw_free) IDL_KWFree();

  return(result);
}
```

New Reentrant API

```
IDL_VPTR IDL_someroutine(int argc, IDL_VPTR *argv, char *argk)
{
  typedef struct {
    IDL_KW_RESULT_FIRST_FIELD; /* Must be first entry in struct */
    IDL_VPTR count_var;
    IDL_LONG debug;
    IDL_STRING name;
  } KW_RESULT;
  static IDL_KW_PAR kw_pars[] = {
    { "COUNT", 0, 1, IDL_KW_OUT | IDL_KW_ZERO,
      0, IDL_KW_OFFSETOF(count_var) },
    { "DEBUG", IDL_TYP_LONG, 1, IDL_KW_ZERO,
      0, IDL_KW_OFFSETOF(debug) },
    { "NAME", IDL_TYP_STRING, 1, IDL_KW_ZERO,
      0, IDL_KW_OFFSETOF(name) },
    { NULL }
  };

  KW_RESULT kw;
  IDL_VPTR result;

  argc = IDL_KWProcessByOffset(argc, argv, argk, kw_pars,
                               (IDL_VPTR *) 0, 1, &kw);

  /* Your code goes here. Keyword values are available in the
   * kw struct.*/

  /* Cleanup keywords before leaving if necessary */
  IDL_KW_FREE;

  return(result);
}
```



Chapter 14: IDL Internals: String Processing

This chapter discusses the following topics:

String Processing and IDL

A number of functions exist to simplify the processing of **IDL_STRING** descriptors. By using these functions instead of doing your own string management, you can eliminate a common source of errors.

Accessing IDL_STRING Values

It is important to realize that the **s** field of an **IDL_STRING** struct does not contain a valid string pointer in the case of a null string (i.e., when **slen** is zero). A common error that can cause IDL to crash is illustrated by the following code fragment:

```
void print_str(IDL_STRING *s)
{
  printf("%s", s->s);
}
```

The problem with this code is that it fails to consider the case where the argument **s** describes a null string. The proper way to write this code is as follows:

```
void print str(IDL_STRING *s)
{
  printf("%s", IDL_STRING_STR(s));
}
```

The macro **IDL_STRING_STR** takes as its argument a pointer to an **IDL_STRING** struct. If the string is null, it returns a pointer to a zero length null-terminated string, otherwise it returns the string pointer from the struct. Consistent use of this macro will avoid the most common sort of error involving strings.

It is common for IDL system routines to accept arguments that provide names. Such arguments must be scalar strings, or string arrays that contain a single element. To properly process such an argument, it is necessary to screen out non-string types or multi-element arrays, locate the string descriptor, and use the **IDL_STRING_STR()** macro to extract a usable NULL terminated C string from it. The **IDL_VarGetString()** is used for this purpose. It encapsulates all of the error checking, and always returns a pointer to a NULL terminated C string, throwing the proper **IDL_MSG_LONGJMP** error via the **IDL_Message()** function when this is not possible:

```
char *IDL_VarGetString(IDL_VPTR v)
```

where

v

```
Variable from which string value is desired.
```

Copying Strings

It is often necessary to copy one string to another. Assume, for example, that there are two string descriptors **s_src** and **s_dst**, and that **s_dst** contains garbage. It would almost suffice to simply copy the contents of **s_src** into **s_dst**. The reason this is not quite correct is that both descriptors would then contain a pointer to the same string. This aliasing can cause some strange effects, or even cause IDL to crash if one of the two descriptors is freed and the string from the other is accessed.

IDL_StrDup() takes care of this problem by allocating memory for a second copy of the string, and replacing the string pointer in the descriptor with a pointer to the fresh copy. Naturally, if the string descriptor is for a null string, nothing is done.

```
void IDL_StrDup(IDL_STRING *str, IDL_MEMINT n)
```

where:

str

Pointer to one or more **IDL_STRING** descriptors which need their strings duplicated.

n

The number of descriptors.

The proper way to copy a string is:

```
s_dst = s_src;            /* Copy the descriptor */
IDL_StrDup(&s_dst, 1L);   /* Duplicate the string */
```

Deleting Strings

Before an **IDL_STRING** can be discarded or re-used, it is important to release any dynamic memory it might be using. The **IDL_StrDelete()** function should be used to delete strings:

```
void IDL_StrDelete(IDL_STRING *str, IDL_MEMINT n)
```

where:

str

Pointer to one or more **IDL_STRING** descriptors which need their contents freed.

n

The number of descriptors.

IDL_StrDelete() deletes all dynamic memory used by the **IDL_STRING**s. The descriptors contain garbage once this has been done, and their contents should not be used.

The **IDL_Deltmp()** function automatically calls **IDL_StrDelete()** when returning temporary variables of type **IDL_TYP_STRING**, so it is not necessary or desirable to call **IDL_StrDelete()** explicitly in this case.

Setting an IDL_STRING Value

The **IDL_StrStore()** function should be used to store a null-terminated C string into an **IDL_STRING** descriptor:

```
void IDL_StrStore(IDL_STRING *s, char *fs)
```

where:

s

Pointer to an **IDL_STRING** descriptor. This descriptor is assumed to contain garbage, so call **IDL_StrDelete()** on it first if this is not the case.

fs

Pointer to the null-terminated string to be copied into s.

IDL_StrStore() is useful for placing a string value into an **IDL_STRING**. This **IDL_STRING** does not need to be a component of a **VARIABLE**, which makes this function very flexible.

One often needs a temporary, scalar **VARIABLE** of type **IDL_TYP_STRING** with a given value. The function **IDL_StrToSTRING()** fills this need:

```
IDL_VPTR IDL_StrToSTRING(char *s)
```

where:

s

Pointer to the null-terminated string to be copied into the resulting temporary variable.

Obtaining a String of a Given Length

Sometimes you need to make sure that the string in an **IDL_STRING** descriptor has a specific length. The **IDL_StrEnsureLength()** function can be used in this case:

```
void IDL_StrEnsureLength(IDL_STRING *s, int n)
```

where:

s

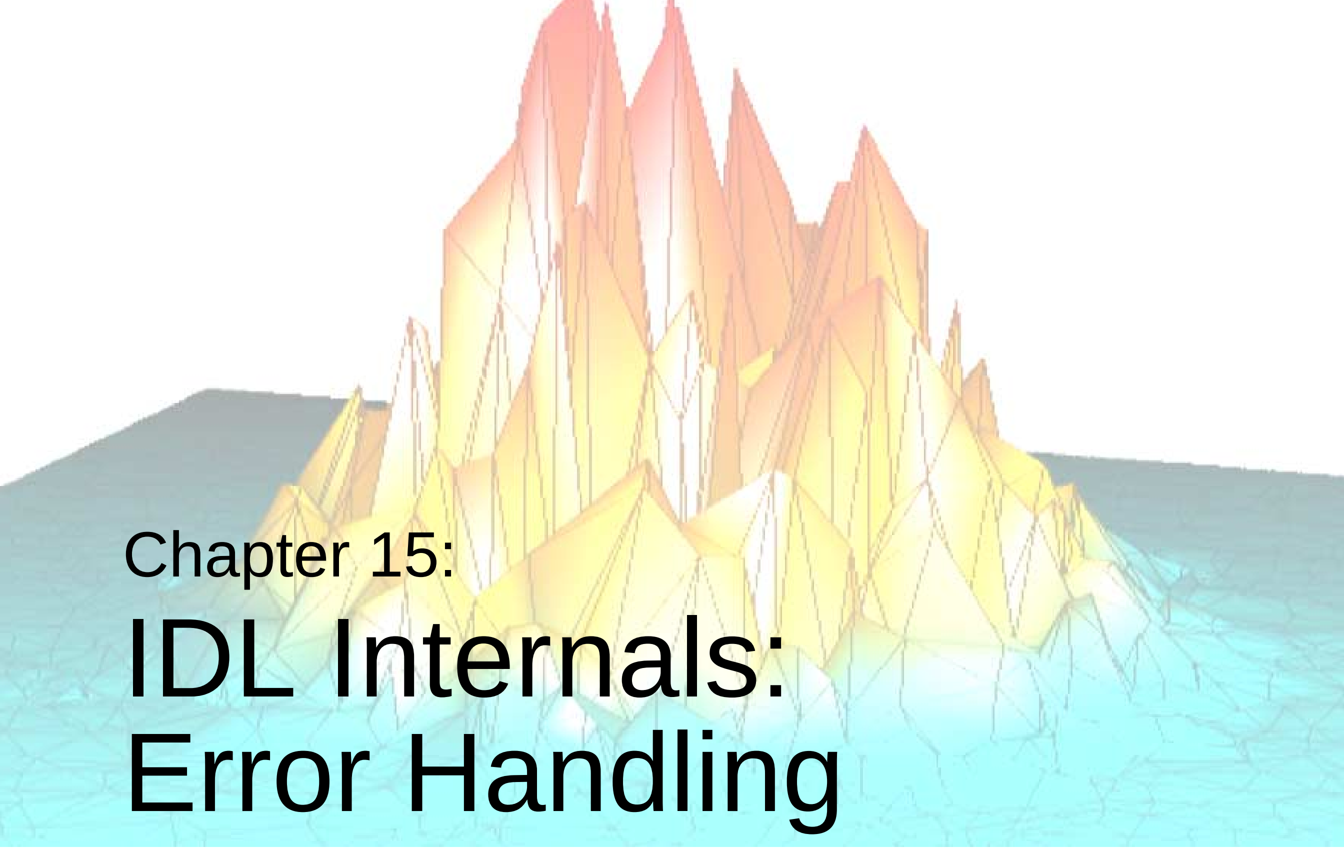
A pointer to the **IDL_STRING** that will have its length checked.

n

The number of characters the string must be able to contain, not including the terminating null character.

If the **IDL_STRING** passed already has enough room for the specified number of characters, it is not re-allocated. Otherwise, the existing string is freed and a new string of sufficient length is allocated. In either case, the **slen** field of the **IDL_STRING** will be set to the requested length.

If a new dynamic string is allocated, it will contain garbage values because **IDL_StrEnsureLength()** only allocates memory of the specified size, it does not copy a value into it. Therefore, the calling routine must copy a null-terminated string into the new dynamic string.



Chapter 15: IDL Internals: Error Handling

This chapter discusses the following topics:

Message Blocks

IDL maintains messages in opaque data structures known as *Message Blocks*. A message block contains all the messages for a logically related area.

When IDL starts, there is only one defined block named **IDL_MBLK_CORE**, containing all messages defined by the core IDL product. Typically, dynamically loadable modules (DLMs) each define a message block for their error messages when they are loaded (See "Dynamically Loadable Modules" on page 450 for a description of DLMs).

There are often two versions of IDL message module functions. Those with names that end in **FromBlock** require an explicit message block. The versions that do not end in **FromBlock** use the **IDL_MBLK_CORE** message block.

To define a message block, you must supply an array of **IDL_MSG_DEF** structures:

```
typedef struct {
  char *name;
  char *format;
} IDL_MSG_DEF;
```

where:

name

A string giving the name of the message. We suggest that you adopt a consistent unique prefix for all your error codes. All message codes defined by RSI start with the prefix **IDL_M_**. You should not use this prefix when naming your blocks in order to avoid unnecessary name collisions.

format

A format string, in printf(3) format. There is one extension to the printf formatting codes: If the first two letters of the format are "%N", then IDL will substitute the name of the currently executing IDL procedure or function (if any) followed by a colon and a space when this message is issued. For example:

```
IDL> print, undefined_var
% PRINT: Variable is undefined: UNDEFINED_VAR.
```

The **IDL_MessageDefineBlock()** function is used to define a new message block:

```
IDL_MSG_BLOCK IDL_MessageDefineBlock
(char *block_name, int n, IDL_MSG_DEF *defs)
```

The arguments to **IDL_MessageDefineBlock()** are as follows:

block_name

Name of the message block. This can be any string, but it will be case folded to upper case. We suggest a single word be used. It is important to pick names that are unlikely to be used by any other application. All blocks defined by RSI start with the prefix **IDL_MBLK_**. You should not use this prefix when naming your blocks in order to avoid unnecessary confusion.

n

# of message definitions pointed at by defs.

defs

An array of message definition structs, each one supplying the name and format string for a message in printf(3) format. The memory used for this array, including the strings it points at, must be in permanently allocated read-only storage. IDL does not copy this memory, but simply uses it in place.

If possible, the new message block is defined and an opaque pointer to it is returned. This pointer must be supplied to subsequent calls to the “FromBlock” message module functions to identify the message block a given error is being issued from. If it is not possible to define the message block, this function returns NULL.

The message functions require a message block pointer and the negative index of the specific message to be issued. Hence, message codes start and zero and grow negatively. For mnemonic convenience, it is standard practice to define preprocessor macros to represent the error codes.

Example: Defining A Message Block

The following code defines a message block named TESTMODULE that contains two messages:

```
static IDL_MSG_DEF msg_arr[] =
{
#define M_TM_INPRO 0
  {  "M_TM_INPRO",   "%NThis is from a loadable module procedure."
},
#define M_TM_INFUN -1
  {  "M_TM_INFUN",   "%NThis is from a loadable module function."
},
};

msg_block = IDL_MessageDefineBlock("Testmodule",
                              sizeof(msg_arr)/sizeof(msg_arr[0]),
                              msg_arr);
```

Issuing Error Messages

Errors are reported using one of the following functions:

- **IDL_Message()**
- **IDL_MessageFromBlock()**
- **IDL_MessageSyscode**()
- **IDL_MessageSyscodeFromBlock**()

These functions are patterned after the standard C library **printf()** function. They are really the same function, differing in which message block the error is issued from (the **FromBlock** versions allow you to specify the block) and their reporting of system errors that might accompany IDL errors (the **Syscode** versions allow you to specify a system error). IDL documentation often refers to **IDL_Message**(). This should be understood to be a generic reference to any of these four functions.

```
void IDL_Message(int code, int action, ...)
void IDL_MessageFromBlock(IDL_MSG_BLOCK block,int code,
                          int action, ...)
void IDL_MessageSyscode(int code, IDL_MSG_SYSCODE_T syscode_type,
                        int syscode, int action, ...)
void IDL_MessageSyscodeFromBlock(IDL_MSG_BLOCK block, int code,
                                 IDL_MSG_SYSCODE_T syscode_type,
                                 int syscode, int action, ...)
```

The arguments to are as follows:

block

Pointer to the IDL message block from which the error should be issued. If block is a NULL pointer, the default IDL core block (**IDL_MBLK_CORE**) is used.

code

This is the error code associated with the error message to be issued. There are two error codes in the default IDL core block (**IDL_MBLK_CORE**) that are available to programmers adding system routines to IDL. The use of these codes is described below. See “IDL_M_GENERIC” on page 342 and “IDL_M_NAMED_GENERIC” on page 342.

Note

For any significant development involving an IDL system routine, RSI recommends your code be packaged as a Dynamically Loadable Module (DLM), and that your DLM define a message block to contain its errors instead of using the GENERIC core block messages.

---

syscode_type

IDL_Message() always issues a single-line error message that describes the problem from IDL's point of view. Often, however, there is an underlying system reason for the error that should also be displayed to give the user a complete picture of what went wrong. For example, the IDL view of the problem might be "Unable to open file," while the underlying system reason for the error is "no such directory." The **IDL_MessageSyscode**() functions allow you to include the relevant system error code, and have it incorporated into the IDL message on a second line of output. There are several different types of system error code that can be specified. The **syscode_type** argument is used to tell **IDL_MessageSyscode**() which type of system error is present:

IDL_MSG_SYSCODE_NONE — Indicates that there is no system error. In this case, the **syscode** argument is ignored, and **IDL_MessageSyscode**() is functionally equivalent to **IDL_Message**().

IDL_MSG_SYSCODE_ERRNO — The UNIX operating system uses a system provided global variable named **errno** for communicating system level errors. Whenever a call to a system function fails, it returns a value of -1, and puts an error code into **errno** that specifies the reason for the failure. Other functions, such as those provided by the standard C library, do not set **errno**. The system documentation (man pages) describes which functions do and do not set **errno**, and the rules for interpreting its value.

The C programming language and UNIX operating system share a common heritage, as C was originally created by its authors as an implementation language for UNIX. Since then, C has found broad acceptance on non-UNIX platforms, bringing along with standard POSIX libraries that provide functionality commonly expected by C programs. Hence, although **errno** is a UNIX concept, non-UNIX C implementations generally provide it as a convenience. Hence, IDL supports **IDL_MSG_SYSCODE_ERRNO** on all platforms.

You should specify **IDL_MSG_SYSCODE_ERRNO** only if you are calling **IDL_MessageSyscode()** as the result of a failed function that is documented to set **errno** on your target platform. Otherwise, **errno** might contain an unrelated garbage value resulting in an incorrect error message. When specifying **IDL_MSG_SYSCODE_ERRNO**, you should supply the current value of **errno** as the **syscode** argument to **IDL_MessageSyscode**().

The Microsoft Windows operating system has **errno** for compatibility with the expectations of C programmers, but typically does not set it. On this operating system, specifying **IDL_MSG_SYSCODE_ERRNO** may have no effect.

IDL_MSG_SYSCODE_WIN (Microsoft Windows Only) — Microsoft Windows system error codes. The value suppled to the **syscode** argument to **IDL_MessageSyscode**() should be a system error code, as returned by the Windows **GetLastError**() system function.

IDL_MSG_SYSCODE_WINSOCK (Microsoft Windows Only) — Microsoft Windows winsock error codes. The value suppled to the **syscode** argument to **IDL_MessageSyscode**() should be a system error code, as returned by the Windows **WSAGetLastError**() system function

syscode

Value of the system error code that should be reported. This argument is ignored if its value is zero (0), or if **syscode_type** is **IDL_MSG_SYSCODE_NONE**. Otherwise, it is interpreted as an error code of the type given by **syscode_type**, and the text of the specified system error will be output along with the IDL message on a separate second line.

action

IDL_Message() can take a number of different actions after issuing the error message. The action to take is specified by the **action** argument:

IDL_MSG_RET

Use this argument to make **IDL_Message()** return to the caller after issuing the error message. In this case, the calling routine can either continue or return to the interpreter as it sees fit.

IDL_MSG_INFO

Use this argument to issue a message that is not an error, but is simply informational in nature. The message is output and **IDL_Message()** returns to the caller. Normally, **IDL_Message()** sets the values of IDL's !ERROR_STATE system variables, but not in this case.

IDL_MSG_EXIT

Use this argument to cause the IDL process to exit after the message is issued. This code should never be used in a system function or procedure—it is intended for use in other sections of the system.

IDL_MSG_LONGJMP

Use this argument to cause **IDL_Message()** to exit directly back to the interpreter after issuing the message. In this case, **IDL_Message()** does not return to its caller. It is an error to use this action code in code not called by the IDL interpreter since the resulting call to **longjmp()** will be invalid.

IDL_MSG_IO_LONGJMP

This action code is exactly like **IDL_MSG_LONGJMP**, except that it is issued in response to an input/output error. This code is only used by the I/O module. User written system routines should use the existing I/O routines, so they do not need to use this action.

In addition, the following modifier codes can be ORed into the action code. They modify the normal behavior of **IDL_Message()**:

IDL_MSG_ATTR_NOPRINT

Suppress the printing of the error message, but do everything else in the normal way.

IDL_MSG_ATTR_MORE

Use paging in the style of the UNIX **more** command to display the output. This option exists primarily for use by the IDL compiler, and is unlikely to be of interest to authors of system routines.

IDL_MSG_ATTR_NOPREFIX

Normally, **IDL_Message()** prefixes the output message with the string contained in IDL's **!MSG_PREFIX** system variable. **IDL_MSG_ATTR_NOPREFIX** suppresses this prefix string.

IDL_MSG_ATTR_QUIET

If the **IDL_MSG_INFO** action has been specified and this bit mask has been included, and the IDL user has IDL's !QUIET system variable, **IDL_Message()** returns without issuing a message.

IDL_MSG_ATTR_NOTRACE

Set this code to inhibit the traceback portion of the error message.

IDL_MSG_ATTR_BELL

Set this code to ring the bell when the message is output.

...

The message format string (specified by the **code** argument) specifies a format string to be used for the error message. This format string is exactly like those used by the standard C library **printf()** function. Any arguments following action are taken to be arguments for this format string.

Error Codes

As mentioned above, RSI has reserved two error codes for use by writers of system routines. They are:

IDL_M_GENERIC

This message code simply specifies a format string of “%s”. The first argument after **action** is taken to be a null-terminated string that is substituted into the format string. For example, the C statement:

```
IDL_Message(IDL_M_GENERIC, IDL_MSG_LONGJMP, "Error! Help!")
```

causes IDL to abort the current routine and issue the message:

```
% Error! Help!
```

IDL_M_NAMED_GENERIC

This message code is exactly like the one above, except that it prints the name of the system routine in front of the error string. For example, assuming that the name of the routine is MY_PROC, the C statement:

```
IDL_Message(IDL_M_NAMED_GENERIC, IDL_MSG_LONGJMP,
            "Error! Help!")
```

causes IDL to interrupt the current routine and issue the message:

```
% MY PROC: Error! Help!
```

Choosing an Error Code

Note

For any significant development involving an IDL system routine, RSI recommends your code be packaged as a Dynamically Loadable Module (DLM), and that your DLM define a message block to contain its errors instead of using the GENERIC messages described here.

The choice of which code to use depends on the context in which the message is issued, but **IDL_M_NAMED_GENERIC** is usually preferred.

If you wish to include arguments into your message string, you should use the **sprintf()** function from the C standard library to format a string into a temporary buffer, and then supply the buffer as the argument to **IDL_Message()**. For example, executing the code:

```
char buf[128];
int unit = 23;

sprintf(buf, "Help! Error number %d.", unit);
IDL_Message(IDL_M_GENERIC, IDL_MSG_LONGJMP, buf);
```

interrupts the current routine and issues the message:

```
% Help! Error number 23.
```

Looking Up A Message Code by Name

Given a message block pointer and the name of a message from that block, the **IDL_MessageNameToCode()** function returns the message code that corresponds to it. This is especially useful for dynamically loadable modules that need to throw errors from the IDL core block. The actual error codes are subject to change between IDL releases, so looking them up this way at run-time allows a given DLM to work with different IDL versions.

```
int IDL_MessageNameToCode(IDL_MSG_BLOCK block, char *name)
```

where:

block

Message block name should be translated against, or NULL to use the default core IDL block.

name

The message name for which the code is desired. Name is case sensitive, and should usually be specified as uppercase.

IDL_MessageNameToCode () returns the message code, or 0 if it is not found.

Checking Arguments

IDL allows a user to provide any number of arguments, of any type, to system functions and procedures. IDL checks for a valid number of arguments, but the routine itself must check the validity of types. This task consists of examining the **argv** argument to the routine checking the type and flags field of each argument for suitability. The **IDL_StoreScalar()** function (see "Storing Scalar Values" on page 292) can be very useful in checking write-only arguments.

A number of macros exist in order to simplify testing of variable attributes. All of these macros accept a single argument—the VPTR to the argument in question. The macros check for a desired condition and use the **IDL_Message()** function with the **IDL_MSG_LONGJMP** action to return to the interpreter if an argument type doesn't agree. Some of these macros overlap, and some are contradictory. You should select the smallest set that covers your requirements for each argument. For an example that uses one of these macros, see "Example: A Complete Numerical Routine Example (FZ_ROOTS2)" on page 420.

IDL_EXCLUDE_UNDEF

The argument must not be of type **IDL_TYP_UNDEF**. This condition is usually imposed if the argument is intended to provide some input information to the routine.

IDL_EXCLUDE_CONST

The argument must not be a constant. This condition should be specified if your routine intends to change the value of the argument.

IDL_EXCLUDE_EXPR

The argument must not be a constant or a temporary variable (i.e., the argument must be a named variable). Specify this condition if you intend to return a value in the argument. Returning a value in a temporary variable is pointless because the interpreter will remove it from the stack as soon as the routine completes, causing it to be freed for re-use.

The **IDL_VarCopy()** and **IDL_StoreScalar()** functions automatically check their destination and issue an error if it is an expression. Therefore, if you are using one of these functions to write the new value into the argument variable, you do not need to perform this check first.

IDL_EXCLUDE_FILE

The argument cannot be a file variable (as returned by the IDL ASSOC) function. Most system routines exclude file variables—they are handled by a small set of existing routines. This check is also handled by the **IDL_ENSURE_SIMPLE** macro, which also excludes structure variables.

IDL_EXCLUDE_STRUCT

The argument cannot be a structure.

IDL_EXCLUDE_COMPLEX

The argument cannot be **IDL_TYP_COMPLEX**.

IDL_EXCLUDE_STRING

The argument cannot be **IDL_TYP_STRING**.

IDL_EXCLUDE_SCALAR

The argument cannot be a scalar.

IDL_ENSURE_ARRAY

The argument must be an array.

IDL_ENSURE_OBJREF

The argument must be an object reference heap variable.

IDL_ENSURE_PTR

The argument must be a pointer heap variable.

IDL_ENSURE_SCALAR

The argument must be a scalar.

IDL_ENSURE_STRING

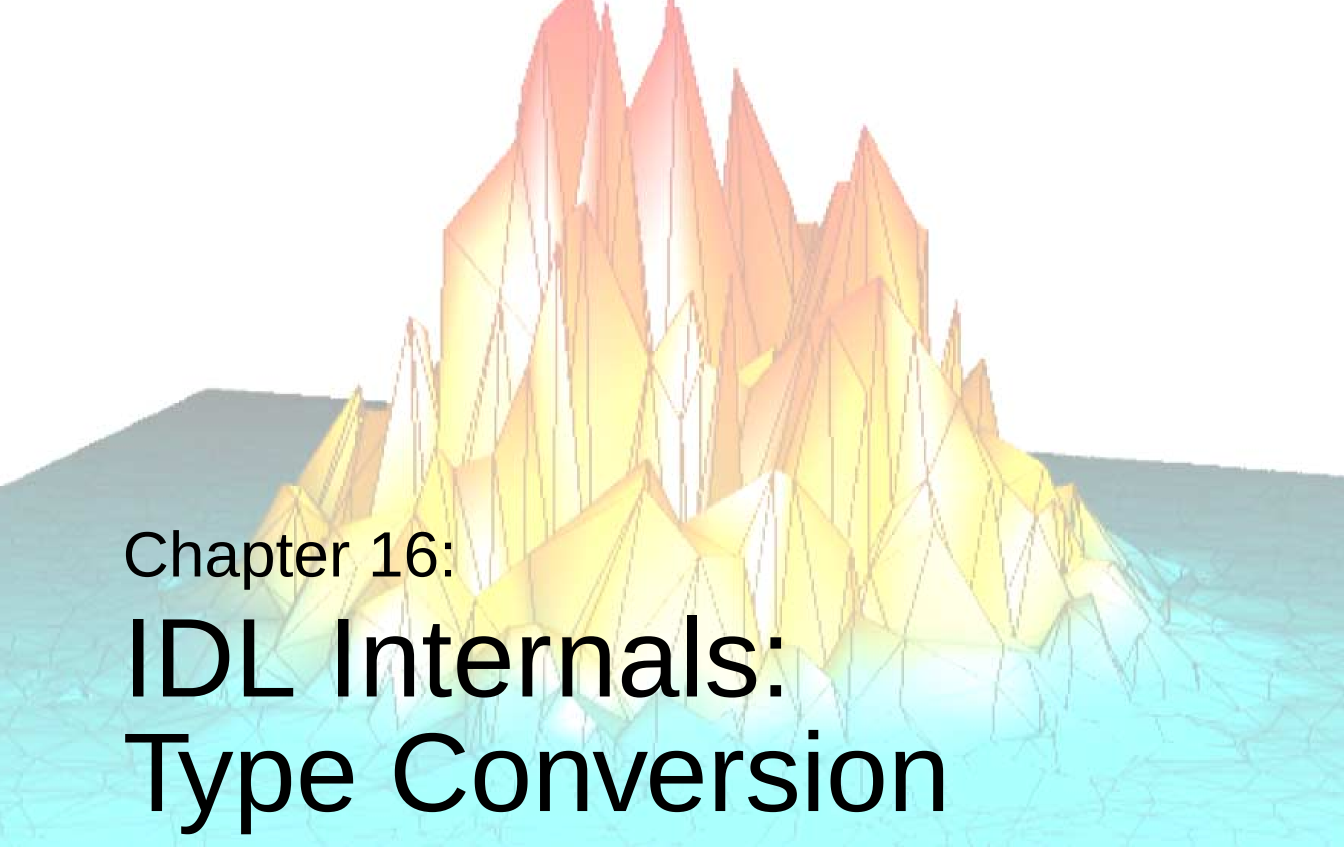
The argument must be **IDL_TYP_STRING**.

IDL_ENSURE_SIMPLE

The argument cannot be a file variable, a structure variable, a pointer heap variable, or an object reference heap variable.

IDL_ENSURE_STRUCTURE

The argument must be **IDL_TYP_STRUCT**.



Chapter 16:

IDL Internals: Type Conversion

This chapter discusses the following topics:

Converting Arguments to C Scalars

The routines described in this section convert the value of their IDL_VARIABLE argument to the C scalar type indicated by their name. **IDL_MEMINTScalar()** and **IDL_FILEINTScalar()** exist for processing memory and file sizes without the need to know their actual types, as discussed in "IDL_MEMINT and IDL_FILEINT Types" on page 263.The converted value is returned as the function value. The functions are defined as:

```
IDL_LONG IDL_LongScalar(IDL_VPTR p)
IDL_ULONG IDL_ULongScalar(IDL_VPTR v)
IDL_LONG64 IDL_Long64Scalar(IDL_VPTR v)
IDL_ULONG64 IDL_ULong64Scalar(IDL_VPTR v)
double IDL_DoubleScalar(IDL_VPTR p)
IDL_MEMINT IDL_MEMINTScalar(IDL_VPTR p)
IDL_FILEINT IDL_FILEINTScalar(IDL_VPTR p)
```

If these functions are unable to perform the conversion (e.g., the argument is a file variable, an array, etc.), they issue a descriptive error and jump back to the interpreter. By using these functions, you avoid having to do any of the type checking described in "Checking Arguments" on page 345.

For example, the following IDL system function (named PRINT_LONG) prints the value of its first argument, converted to an IDL_LONG 32-bit integer:

```
IDL_VPTR print_long(int argc, IDL_VPTR argv[], char *argk)
{
  printf("%d\n", IDL_LongScalar(argv[0]));
}
```

Executing it as:

```
PRINT_LONG, 23D
```

gives the output:

```
23
```

as expected, while the statement:

```
PRINT_LONG, FINDGEN(10)
```

causes the error:

```
% PRINT_LONG: Expression must be a scalar in this context:
              <FLOAT Array(10)>
% Execution halted at $MAIN$ .
```

because it is not possible to convert an array (the result of FINDGEN) to a scalar.

General Type Conversion

The **IDL_BasicTypeConversion()** function provides general purpose type conversion:

```
IDL_VPTR IDL_BasicTypeConversion(int argc, IDL_VPTR argv[]
                                 int type)
```

where:

argc

The number of **IDL_VPTR**s contained in **argv**.

argv

An array of pointers to **VARIABLE** arguments.

type

The desired type code of the result. See "Type Codes" on page 258.

If **argc** is 1, this function returns a pointer to a temporary **VARIABLE** containing the value of **argv[0]** converted to the type specified by the **type** argument. If the variable is already of the correct type, the variable itself is returned.

If **argv** is greater than 1, **argv[1]** is taken to be an offset into the variable specified by **argv[0]**, and following arguments are taken as the dimensions to be used for the result. In this case, enough bytes are copied (starting from the offset) to satisfy the requirements of the dimensions given. This second form does not work for variables of type string, so an error is issued in that case. RSI recommends ensuring that variables of appropriate type are used with this function.

The IDL BYTE and STRING system routines (implemented by the **IDL_CvtByte()** and **IDL_CvtString()** functions, described below) treat conversions between variables of type byte and string in a special way. **IDL_BasicTypeConversion()** does not handle this special case. Instead, it simply performs a straightforward type conversion between those types.

Converting to Specific Types

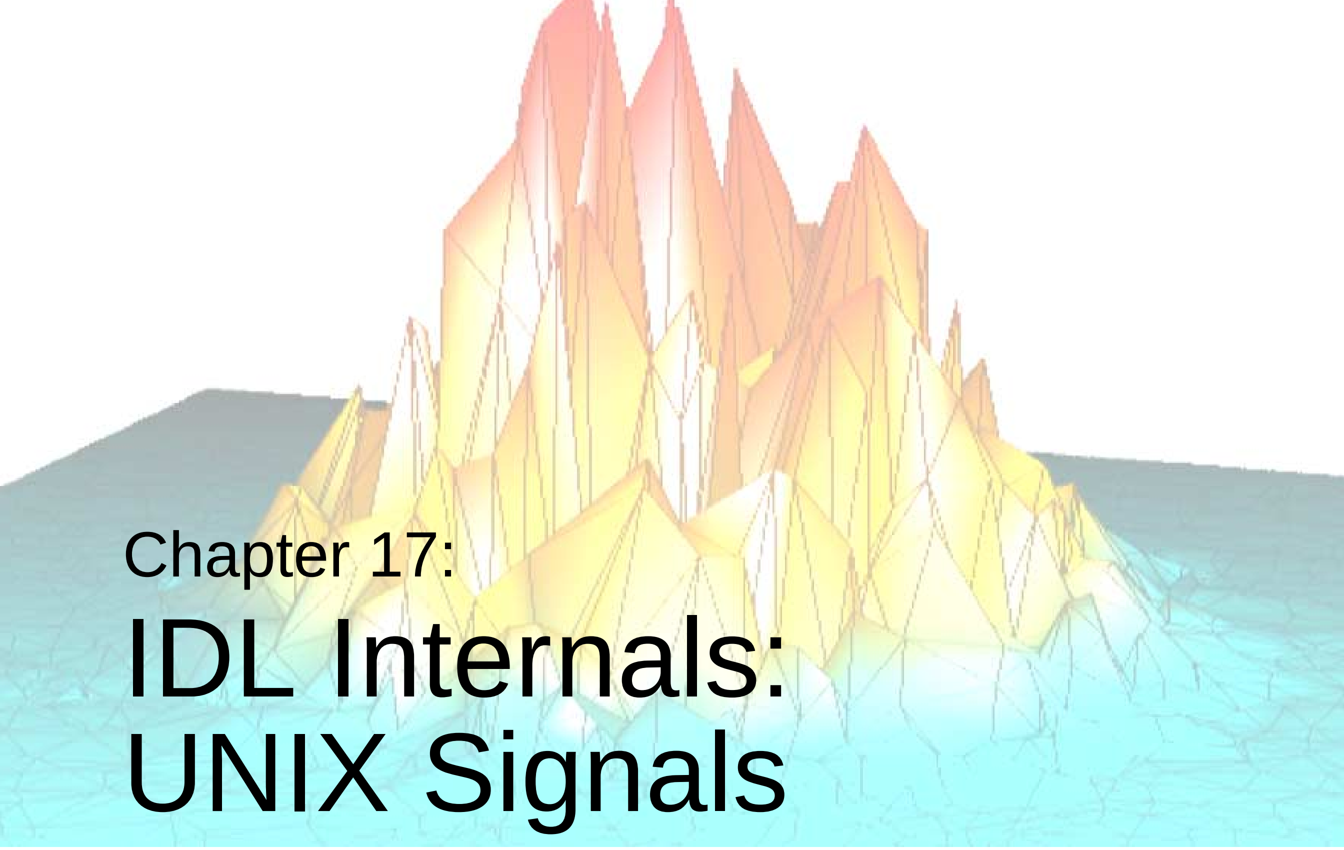
A series of functions exist to convert **VARIABLEs** to specific types:

```
IDL_VPTR IDL_CvtByte(int argc, IDL_VPTR argv[])
IDL_VPTR IDL_CvtBytscl(int argc, IDL_VPTR argv[], char *argk)
IDL_VPTR IDL_CvtFix(int argc, IDL_VPTR argv[])
IDL_VPTR IDL_CvtUInt(int argc, IDL_VPTR argv[])
IDL_VPTR IDL_CvtLng(int argc, IDL_VPTR argv[])
IDL_VPTR IDL_CvtULng(int argc, IDL_VPTR argv[])
IDL_VPTR IDL_CvtLng64(int argc, IDL_VPTR argv[])
IDL_VPTR IDL_CvtULng64(int argc, IDL_VPTR argv[])
IDL_VPTR IDL_CvtFlt(int argc, IDL_VPTR argv[])
IDL_VPTR IDL_CvtDbl(int argc, IDL_VPTR argv[])
IDL_VPTR IDL_CvtComplex(int argc, IDL_VPTR argv[])
IDL_VPTR IDL_CvtDComplex(int argc, IDL_VPTR argv[])
IDL_VPTR IDL_CvtString(int argc, IDL_VPTR argv[], char *argk)
```

When calling these functions, you should set the **argk** argument to NULL.

These functions are the direct implementations of the IDL commands BYTE, BYTSCL, FIX, UINT, LONG, ULONG, LONG64, ULONG64, FLOAT, DOUBLE, COMPLEX, DCOMPLEX, and STRING. See the description of these functions in the *IDL Reference Guide* for details on their arguments and calling sequences.

The behavior of these functions is the same as **IDL_BasicTypeConversion()** except when converting between bytes and strings. Calling **IDL_CvtByte()** with a single argument of string type causes each string to be converted to a byte vector of the same length as the string. Each array element is the character code of the corresponding character in the string. Calling **IDL_CvtString()** with a single argument of IDL_TYP_BYTE has the opposite effect.



Chapter 17: IDL Internals: UNIX Signals

This chapter discusses the following topics:

IDL and Signals

Signals pose one of the more difficult challenges faced by the UNIX programmer. Although seemingly simple, they are a tough portability problem because there are several variants, and their semantics are subtle, inconvenient, and easily confused. These issues are only magnified when signals are used in a program like IDL that employs multiple threads. IDL has always done whatever is necessary with signals in order to get its job done, but its signal assumptions can also affect user written code linked to it.

Note

This discussion refers primarily to UNIX IDL. Microsoft Windows uses different mechanisms to solve the problems solved by signals under UNIX.

The following is a brief list of problems and contradictions inherent in UNIX signals. For a more complete description, see Chapter 10 of *External Programming in the UNIX Environment* by W. Richard Stevens.

- POSIX signals (sigaction) promise to unify and simplify signals, but not all platforms support them fully.
- You can only have one signal handler function registered for each signal. This means that if one part of a program uses a signal, the rest of the program must leave that signal alone.
- In order to meet the needs of programs originally developed under different UNIX systems (AT&T System V, BSD, Posix), most UNIX implementations provide more than one package of signal functions. Typically, a given program is restricted to one of these libraries. If a programmer links code into IDL that chooses a library or signal options different from that used by IDL itself, unexpected results may occur.
- The number and exact semantics of some signals differ in different versions.
- Details of signal blocking differ.
- Some System V implementations of signals are unreliable, meaning that signals can occur in a process and be missed.
- Some older System V systems reset the handling action to **SIG_DFL** before calling the handler. This opens a window in time where two signals in a row can cause the process to be killed. Also, the signal handler must re-establish itself every time it is called.

- On most platforms, if a signal is generated more than once while it is blocked, the second and subsequent occurrences are lost. In other words, most UNIX implementations do not queue signals.

These are among the reasons that most libraries avoid signals, and leave their use to the end programmer. IDL, however, must use signals to function properly. In order to allow users to link their code into IDL while using signals, IDL provides a signal API built on top of the signal mechanism supported by the target platform. The IDL signal API has the following attributes:

- It disallows use of **SIGTRAP** and **SIGFPE**. These signals are reserved to IDL.
- It disallows use of **SIGALRM**. Most uses for **SIGALRM** are provided by the IDL timer API.
- It works with all other signals, including those IDL doesn’t currently use, so the interface won’t change over time.
- It allows multiple signal handlers for each signal, so IDL and other code can use the same signal simultaneously.
- It unifies the signal interface by supplying a stable consistent interface with known behavior to the underlying system signal mechanism.
- It keeps IDL in charge of which signal package is used and how.

This is not a perfect solution, it is a compromise between the needs of IDL and programmers wishing to link code with it. Usually, the IDL signal abstraction is sufficient, but it does have the following limitations:

- The calling program must not attempt to catch **SIGTRAP** or **SIGFPE**, either directly or through library routines that use these signals to achieve their ends. Furthermore, the IDL signal abstraction does not allow the caller to catch these signals, so your program must leave exception handling to IDL.
- The caller loses control over signal package choice and some minor signal abilities.
- Having multiple signal handler routines for a given signal opens the possibility that one handler might do something that causes problems for the others (like change the signal mask, or longjmp()). To minimize such problems, user code linked into IDL must not call the actual system signal routines, and must not longjmp() out of signal handlers—a tactic that is usually allowed, but which would seriously damage IDL’s signal state.
- Since there may be more than one signal handler registered for a given signal, the signal dispositions of **SIG_IGN** and **SIG_DFL** are not directly available to

the caller as they would be if you were allowed to use the system signal facilities directly.

If you find that these restrictions are too limiting for your application, chances are your code is not compatible with IDL and should be executed in a separate process. We then encourage you to consider running IDL in a separate process and to use an interprocess communication mechanism such as RPC.

Signal Handlers

IDL signal handler functions are defined as:

```
typedef void (* IDL_SignalHandler_t)(int signo);
```

When a signal is delivered to the process, all registered signal handlers are called. `signo` is the integer number of the signal delivered, as defined by the C language header file `signal.h` (found in `/usr/include/signal.h` on most UNIX systems). `signo` can be used by a signal handler registered for more than one signal to tell which signal called it.

Establishing a Signal Handler

To register a signal handler, use the **IDL_SignalRegister()** function:

```
int IDL_SignalRegister(int signo, IDL_SignalHandler_t func,
                       int msg_action)
```

where:

signo

The numeric value of the signal to register for, as defined in `signal.h`.

func

The signal handler to be called when the signal specified by `signo` is raised.

msg_action

One of the **IDL_MSG_*** action codes for **IDL_Message()**. If there is an error in registering the signal handler, this action code is passed to **IDL_Message()** to direct its recovery action. Note that it is incorrect to use any of the message codes that cause **IDL_Message()** to **longjmp()** back to the IDL interpreter if your code is running in a context where the IDL interpreter is not active—specifically as part of using Callable IDL.

If `func` is successfully registered for `signo`, this routine returns TRUE. Otherwise, FALSE is returned and **IDL_Message()** is called with `msg_action` to control its behavior. Note that there are values of `msg_action` that result in this routine not returning on error. Multiple registration of the same function is allowed, but has no additional effect—the handler will only be called once.

Removing a Signal Handler

To remove a signal handler, use the **IDL_SignalUnregister()** function:

```
export int IDL_SignalUnregister(int signo, IDL_SignalHandler_t func,
   int msg_action)
```

where:

signo

The signal to unregister.

func

The handler to be unregistered.

msg_action

One of the **IDL_MSG_*** action codes for **IDL_Message()**. If there is an error in removing the signal handler, this action code is passed to **IDL_Message()** to direct its recovery action.

Once **IDL_SignalUnregister()** has been called, **func** is unregistered and will no longer be called if the signal is raised. **IDL_SignalUnregister()** returns TRUE for success, FALSE for failure. Requests to unregister a function that has not been previously registered are ignored.

UNIX Signal Masks

UNIX processes contain a signal mask that defines which signals can be delivered and which are blocked from delivery at any given time. When a signal arrives, the UNIX kernel checks the signal mask: If the signal is in the process mask, it is delivered, otherwise it is noted as undeliverable and nothing further is done until the signal mask changes. Sets of signals are represented within IDL with the opaque type **IDL_SignalSet_t**. UNIX IDL provides several functions that manipulate signal sets to change the process mask and allow/disallow delivery of signals.

IDL_SignalSetInit()

IDL_SignalSetInit() initializes a signal set to be empty, and optionally sets it to contain one signal.

```
void IDL_SignalSetInit(IDL_SignalSet_t *set, int signo)
```

where:

set

The signal set to be emptied/initialized.

signo

If non-zero, a signal to be added to the new set. This is provided as a convenience for the large number of cases where a set contains only one signal. Use **IDL_SignalSetAdd()** to add additional signals to a set.

IDL_SignalSetAdd()

IDL_SignalSetAdd() adds the specified signal to the specified signal set:

```
void IDL_SignalSetAdd(IDL_SignalSet_t *set, int signo)
```

where:

set

The signal set to be added to. The signal set must have been initialized by **IDL_SignalSetInit()**.

signo

The signal to be added to the signal set.

IDL_SignalSetDel()

IDL_SignalSetDel() deletes the specified signal from a signal set:

```
void IDL_SignalSetDel(IDL_SignalSet_t *set, int signo)
```

where:

set

The signal set to delete from. The signal set must have been initialized by **IDL_SignalSetInit()**.

signo

The signal to be removed from the signal set.

IDL_SignalSetIsMember()

IDL_SignalSetIsMember() tests a signal set for the presence of a specified signal, returning TRUE if the signal is present and FALSE otherwise:

```
int IDL_SignalSetIsMember(IDL_SignalSet_t *set, int signo)
```

where:

set

The signal set to test. The signal set must have been initialized by **IDL_SignalSetInit()**.

signo

The signal to be removed from the signal set.

IDL_SignalMaskGet()

IDL_SignalMaskGet() sets a signal set to contain the signals from the current process signal mask:

```
void IDL_SignalMaskGet(IDL_SignalSet_t *set)
```

where:

set

The signal set in which the current process signal mask will be stored.

IDL_SignalMaskSet()

IDL_SignalMaskSet() sets the current process signal mask to contain the signals specified in a signal mask:

```
void IDL_SignalMaskSet(IDL_SignalSet_t *set,
                       IDL_SignalSet_t *omask)
```

where:

set

The signal set from which the current process signal mask will be set.

omask

If **omask** is non-NULL, the unmodified process signal mask is stored in it. This is useful for restoring the mask later using **IDL_SignalMaskSet()**.

There are some signals that cannot be blocked. This limitation is silently enforced by the operating system.

IDL_SignalMaskBlock()

IDL_SignalMaskBlock() adds signals to the current process signal mask:

```
void IDL_SignalMaskBlock(IDL_SignalSet_t *set,
                         IDL_SignalSet_t *oset)
```

where:

set

The signal set containing the signals that will be added to the current process signal mask.

oset

If **oset** is non-NULL, the unmodified process signal mask is stored in it. This is useful for restoring the mask later using **IDL_SignalMaskSet()**.

There are some signals that cannot be blocked. This limitation is silently enforced by the operating system.

IDL_SignalBlock()

IDL_SignalBlock() does the same thing as **IDL_SignalMaskBlock()** except it accepts a single signal number instead of requiring a mask to be built:

```
void IDL_SignalBlock(int signo, IDL_SignalSet_t *oset)
```

where:

signo

The signal to be blocked.

There are some signals that cannot be blocked. This limitation is silently enforced by the operating system.

IDL_SignalSuspend()

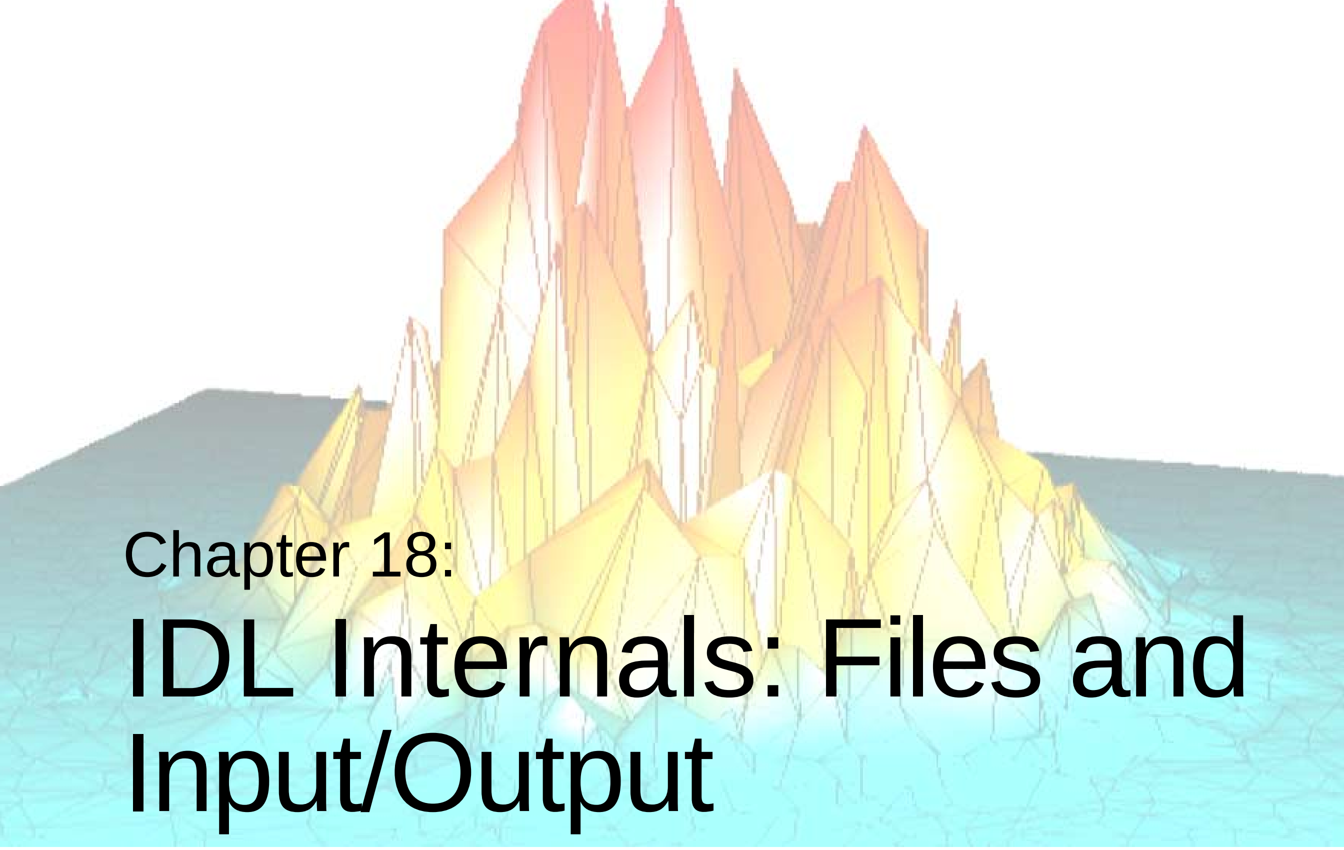
IDL_SignalSuspend() replaces the process signal mask with the ones in set and then suspends the process until a signal is delivered. On return, the original process signal mask is restored:

```
void IDL_SignalSuspend(IDL_SignalSet_t *set)
```

where:

set

The signal set containing the signals that will be added to the current process signal mask.



Chapter 18: IDL Internals: Files and Input/Output

This chapter discusses the following topics:

IDL and Input/Output Files

IDL provides extensive Input/Output facilities at the user level. Internally, it uses native Input/Output facilities (UNIX system calls or Win32 system API) in addition to the standard C library stream package (stdio). The choice of which facilities to use are made based on the target platform and the requested features for the file.

Most external code linked with IDL (CALL_EXTERNAL, system routines, etc.) should not do Input/Output directly, for the following reasons:

- Part of the IDL philosophy is that Input/Output is handled by dedicated I/O facilities provided by IDL, and that computational code should accept data from IDL variables and return results in the same way. This gives the user of your code the freedom and flexibility to save their data in any of the many forms supported by IDL’s core I/O facilities, and frees you from writing complex and error prone input/output code.
- Using IDL’s Input/Output facilities frees you from having to code around platform specific differences in I/O behavior.
- Input/Output from languages other than C often require runtime library support code to run at program startup before your code and successfully perform I/O. For example, Fortran Input/Output may depend on a Fortran runtime subsystem having been initialized. IDL, as a C program, does not perform initialization of such libraries for other languages. If you know enough about your Fortran system, you can often supply the missing initialization call, but such workarounds are usually not well documented, and are inherently platform specific.

For the reasons above, only minimal I/O abilities are available from IDL's internals, and only for files that explicitly use the standard C stdio library. Therefore, if your application must directly perform I/O to a file managed by IDL, it is necessary to use the standard C library streampackage (stdio) by specifying the IDL_F_STDIO flag to **IDL_FileOpen()**. Most of the routines associated with the standard C library I/O package can be used in the normal manner. Note, however, that the C library routines listed in the following table should not be used; use the IDL-specific functions instead:

C Library Function	IDL Function
fclose()	IDL_FileClose()
fdopen()	IDL_FileOpen()
feof()	IDL_FileEOF()
fflush()	IDL_FileFlushUnit()
fopen()	IDL_FileOpen()
freopen()	IDL_FileOpen()

Table 18-1: Disallowed C Library Routines and Their IDL Counterparts

Note

In order to access a file opened using IDL_FileOpen() in this manner, you must ensure that it is stdio compatible by specifying IDL_F_STDIO as part of the extra_flags argument to IDL_FileOpen(). Failure to do this will cause your code to fail to execute as expected.

File Information

IDL maintains a file table in which it keeps a file descriptor for each file opened with IDL_FileOpen(). This table is indexed by the file Logical Unit Number, or LUN. These are the same LUNs IDL users use.

The IDL_FileStat() function is used to get information about a file.

IDL_FileStat()

```
void IDL_FileStat(int unit, IDL_FILE_STAT *stat_blk)
```

unit

The logical unit number (LUN) of the file unit to be checked. This function should only be called on file units that are known to be open.

stat_blk

A pointer to an IDL_FILE_STAT struct to be filled in with information about the file. The information returned is valid only as long as the file remains open. You must not access the fields of an IDL_FILE_STAT once the file it refers to has been closed. This struct has the definition:

```
typedef struct {
  char *name;
  short access;
  IDL_SFILE_FLAGS_T flags;
  FILE *fptr;
} IDL_FILE_STAT;
```

The fields of this struct are listed below:

name

A pointer to a null-terminated string containing the name the file was opened with.

access

A bit mask describing the access allowed to the file. The allowed bit values are listed in the following table:

Bit Value	Description
IDL_OPEN_R	The file is open for input.
IDL_OPEN_W	The file is open for output.
IDL_OPEN_TRUNC	The file was truncated when it was opened. This implies that IDL_OPEN_W is also set.
IDL_OPEN_APND	The file was opened with the file pointer set just past the last byte of data in the file (the file is open for appending).

Table 18-2: Bit values for the access field

flags

A bit mask that gives special information about the file. The defined bits are listed in the following table:

Bit Value	Description
IDL_F_ISATTY	The file is a terminal.
IDL_F_ISAGUI	The file is a Graphical User Interface.
IDL_F_NOCLOSE	The CLOSE command will refuse to close the file.
IDL_F_MORE	If the file is a terminal, output is sent through a pager similar to the UNIX `more` command. Details on this pager are not included in this document, and it is therefore not available for general use.
IDL_F_XDR	The file is read/written using XDR (eXternal Data Representation).
IDL_F_DEL_ON_CLOSE	The file will be deleted when it is closed.

Table 18-3: Bit values for the flags field

Bit Value	Description
IDL_F_SR	The file is a SAVE/RESTORE file.
IDL_F_SWAP_ENDIAN	The file has opposite byte order than that of the current system.
IDL_F_VAX_FLOAT	Binary float and double are in VAX F and D format.
IDL_F_COMPRESS	The file is in compressed gzip format. If IDL_F_SR is set (the file is a SAVE/RESTORE file), the file contains zlib compressed data.
IDL_F_UNIX_F77	The file is read/written in the format used by the UNIX Fortran (f77) compiler for unformatted binary data.
IDL_F_UNIX_PIPE	The file is a bi-directional data path connecting IDL to a child process created by the SPAWN procedure.
IDL_F_UNIX_RAWIO (formerly called IDL_F_UNIX_NOSTDIO)	No application level buffering will be performed for the file and all data transfers will go directly to the operating system for processing (e.g. read() and write() system calls under UNIX, Win32 API for MS Windows). Note that setting this bit does not guarantee that data will be written to the file immediately, because the operating system may buffer the data. This bit value was formerly called IDL_F_UNIX_NOSTDIO. IDL_F_UNIX_RAWIO is the preferred form, but both names are supported.
IDL_F_UNIX_SPECIAL	The file is a UNIX device special file, most likely a pipe. This differs from IDL_F_UNIX_PIPE because it applies to any file, not only those opened with the SPAWN procedure.

Table 18-3: (Continued) Bit values for the flags field (Continued)

Bit Value	Description
IDL_F_STDIO	Use the C standard I/O library (stdio) to perform I/O on this file instead of any other native OS API that might be otherwise used. People intending to access IDL files via their own code should specify this flag if they intend to access the file from their external code as a stdio stream.
IDL_F_SOCKET	File is an internet TCP/IP socket.

Table 18-3: (Continued) Bit values for the flags field (Continued)

fptr

The stdio stream file pointer to the file. This field can be used with standard library functions to perform I/O. This field is always valid, although you shouldn't use it if the file is an XDR file. You can check for this by looking for the **IDL_F_XDR** bit in the flags field.

If the file is not opened with the **IDL_F_STDIO** flag, **fptr** may be returned as an unusable NULL pointer, reflecting the fact that IDL is not using stdio to perform I/O on the file. If access to a valid **fptr** is important to your application, you should be sure to specify **IDL_F_STDIO** to the **extra_flags** argument to **IDL_FileOpen**, or use the STDIO keyword to OPEN if opening the file from the IDL user level.

In addition to the requirement to set the **IDL_F_STDIO** flag, you should be aware that IDL buffers I/O at a layer above the stdio package. If your code does I/O directly to a file that is also being written to from the IDL user level, the IDL buffer may cause data to be written to the file in a different order than you expect. There are several approaches you can take to prevent this:

- Tell IDL not to buffer, by opening the file from the IDL user level and specifying a value of -1 to the BUFSIZE keyword.
- Disable stdio buffering by calling the stdio **setbuf**() function.
- Ensure that you flush IDL's buffer before you do any Input/Output, as discussed in "Flushing Buffered Data" on page 382.

Opening Files

Files are opened using the IDL_FileOpen() function.

IDL_FileOpen()

```
int IDL_FileOpen(int argc, IDL_VPTR *argv, char *argk,
    int access_mode, IDL_SFILE_FLAGS_T extra_flags,
    int longjmp_safe, int msg_attr)
```

IDL_FileOpen() returns TRUE if the file has been successfully opened and FALSE otherwise.

Note

If longjmp_safe is TRUE, the usual course is to jump back to the IDL interpreter, in which case the routine won't return.

argc

The number of arguments in *argv*. This value should always be 2.

argv

The arguments to IDL_File_Open(). argv[0] should be a scalar integer value giving the file unit number (LUN) to be opened. argv[1] is a scalar string giving the file name.

argk

Keywords. Set this argument to NULL.

access_mode

A bit mask that specifies the access to be allowed to the file being opened. The allowed bit values are listed in the following table:

Bit Value	Description
IDL_OPEN_R	The file is open for input.
IDL_OPEN_W	The file is open for output.

Table 18-4: Bit Values for the access_mode Argument

Bit Value	Description
IDL_OPEN_TRUNC	The file was truncated when it was opened. This implies that IDL_OPEN_W is also set.
IDL_OPEN_APND	The file was opened with the file pointer set just past the last byte of data in the file (the file is open for appending).

Table 18-4: Bit Values for the access_mode Argument (Continued)

It is important that conflicting bits not be set together (for example, do not specify IDL_OPEN_TRUNC | IDL_OPEN_APND). Also, one or both of IDL_OPEN_READ and IDL_OPEN_WRITE must always be specified.

extra_flags

Used to specify additional file attributes using the flags defined in the description of the flags field of the IDL_FILE_STAT struct (see "File Information" on page 368). Note that some flags are set by IDL based on the actual attributes of the opened file (e.g. IDL_F_ISTTY) and that it makes no sense to set such flags in this mask.

If you intend to use the opened file as a C standard I/O (stdio) stream file, you must specify the **IDL_F_STDIO** flag when calling **IDL_FileOpen()**. Otherwise, IDL may choose not to use stdio.

longjmp_safe

If set to TRUE, **IDL_FileOpen()** is being called in a context where an IDL_MSG_LONGJMP **IDL_Message** action code is okay. If set to FALSE, the routine won't `longjmp()`.

IDL_FileOpen() returns TRUE if the file has been successfully opened and FALSE otherwise. Of course, if longjmp_safe is TRUE, the usual course is to jump back to the IDL interpreter, in which case the routine won't return.

msg_attr

A zero (0), or any combination of the IDL_MSG_ATTR_ flags, used to fine tune the error handling specified by the `longjmp_safe` argument. Note that you must not specify any of the base IDL_MSG_ codes, but only the attributes. The base code (e.g. IDL_MSG_LONGJMP) is determined by the value of `longjmp_safe`. For a discussion of the IDL_MSG_ATTR_ flags, see "Issuing Error Messages" on page 338.

Special File Units

There are three files that are always open. The three units are:

- **IDL_STDIN_UNIT** — Unit 0 (zero) is the standard input for the IDL process.
- **IDL_STDOUT_UNIT** — Unit –1 is the standard output.
- **IDL_STDERR_UNIT** — Unit –2 is the standard error.

Note

The constant IDL_NON_UNIT always has a value that is *not* a valid file unit.

Closing Files

Files are closed using the IDL_FileClose() function.

IDL_FileClose()

```
void IDL_FileClose(int argc, IDL_VPTR *argv, char *argk)
```

argc

The number of arguments in *argv*.

argv

The arguments to the close function. These should be scalar integer values giving the Logical Unit Numbers of the file units to close.

argk

Keywords. Set this argument to NULL.

Preventing File Closing

Use the **IDL_FileSetClose()** function to prevent files from closing. It does this by setting or clearing the IDL_F_NOCLOSE bit in the flags field of the internal file descriptor maintained by IDL for the file (see “File Information” on page 368). This feature is used primarily in graphics drivers for printers. For example, the PostScript driver uses this feature to prevent the user from closing the plot data file prematurely.

When IDL exits, it only closes open files that do not have the IDL_F_NOCLOSE bit set. Files with close inhibited are simply left alone. Often, you will want to declare an exit handler which takes care of closing such files.

IDL_FileSetClose()

```
void IDL_FileSetClose(int unit, int allow)
```

unit

The Logical Unit Number (LUN) of the file in question. The file must be open for this function to have effect.

allow

Set this field to TRUE if users are allowed to close the file. Set to FALSE if users should be prevented from closing the file.

Checking File Status

System routines that deal with files must verify that the files have the proper attributes for the intended operation. Use the function IDL_FileEnsureStatus() for this.

IDL_FileEnsureStatus()

```
int IDL_FileEnsureStatus(int action, int unit, int flags)
```

action

If the file unit does not satisfy the requirements of the flags argument, IDL_FileEnsureStatus() will issue an error using the IDL_Message() function (see “Issuing Error Messages” on page 338). This action is the action argument to IDL_Message() and should be IDL_MSG_RET, IDL_MSG_LONGJMP, or IDL_MSG_IO_LONGJMP.

unit

The Logical Unit Number of the file to be checked.

flags

IDL_FileEnsureStatus() always checks to make sure unit is a valid logical file unit. In addition, flags is a bit mask specifying the file attributes that should be checked. The possible bit values are listed in the following table:

Bit Value	Description
IDL_EFS_USER	The file must be a user unit. This means that the file is not one of the three special files, stdin, stdout, or stderr.
IDL_EFS_IDL_OPEN	The file unit must be open.
IDL_EFS_CLOSED	The file unit must be closed.
IDL_EFS_READ	The file unit must be open for input.
IDL_EFS_WRITE	The file unit must be open for output.
IDL_EFS_NOTTY	The file unit cannot be a tty.

Table 18-5: Bit Values for the flags Argument

Bit Value	Description
IDL_EFS_NOGUI	The file unit cannot be a Graphical User Interface.
IDL_EFS_NOPIPE	The file unit cannot be a pipe.
IDL_EFS_NOXDR	The file unit cannot be a XDR file.
IDL_EFS_ASSOC	The file unit can be ASSOC'ed. This implies IDL_EFS_USER, IDL_EFS_OPEN, IDL_EFS_NOTTY, IDL_EFS_NOPIPE, IDL_EFS_NOXDR, IDL_EFS_NOCOMPRESS, and IDL_EFS_NOSOCKET.
IDL_EFS_NOT_RAWIO (formerly called IDL_EFS_NOT_NOSTDIO)	The file was not opened with the IDL_F_UNIX_RAWIO attribute. This bit was formerly called IDL_F_NOTSTDIO. IDL_EFS_NOT_RAWIO is the preferred form, but both names are accepted.
IDL_EFS_NOCOMPRESS	The file unit cannot have been opened for compressed input/output (IDL_F_COMPRESS).
IDL_EFS_STDIO	The file must be using the C stdio package (IDL_F_STDIO).
IDL_EFS_NOSOCKET	The file unit cannot be a socket (IDL_F_SOCKET).

Table 18-5: (Continued) Bit Values for the flags Argument (Continued)

Note

Some of these values are contradictory. The caller must select a consistent set.

If the file unit meets the desired conditions, IDL_FileEnsureStatus() returns TRUE. If it does not meet the conditions, and action was IDL_MSG_RET, then it returns FALSE.

Allocating and Freeing File Units

System routines must allocate and deallocate file units in order to avoid conflicts. When writing IDL procedures, the GET_LUN and FREE_LUN procedures are used. When writing system-level routines, you can access the same routines by calling IDL_FileGetUnit() and IDL_FileFreeUnit().

Use IDL_FileGetUnit() to allocate file units:

IDL_FileGetUnit()

```
void IDL_FileGetUnit(int argc, IDL_VPTR *argv)
```

argc

argc should always be 1.

argv

argv[0] contains an IDL_VPTR to the IDL_VARIABLE that will be filled in with the resulting unit number.

Use IDL_FileFreeUnit() to free file units:

IDL_FileFreeUnit()

```
void IDL_FileFreeUnit(int argc, IDL_VPTR *argv)
```

argc

argc gives the number of elements in **argv**.

argv

argv should contain scalar integer values giving the Logical Unit Numbers of the file units to be returned.

Read the description of GET_LUN and FREE_LUN in the *IDL Reference Guide* for additional details about these functions. The following code fragment demonstrates how these functions might be used to open and close a file named `junk.dat`:

```
IDL_VPTR argv[2];
IDL_VARIABLE unit;
IDL_VARIABLE name;
:
:
```

```
/* Allocate a file unit. */
argv[0] = &unit;
unit.type = IDL_TYP_LONG;
unit.flags = 0;
IDL_FileGetUnit(1, argv);

/* Set up the file name */
name.type = IDL_TYP_STRING;
name.flags = IDL_V_CONST;
name.value.str.s = "junk.dat";
name.value.str.slen = sizeof("junk.dat") - 1;
name.value.str.stype = 0;
argv[1] = &name;
.
.
.
IDL_FileOpen(2, argv, (char *) 0, IDL_OPEN_R, 0, 1);

/* Perform any required actions. */
.
.
.
/* Free the file unit. This will also close the file. */
IDL_FileFreeUnit(1, argv);
```

Detecting End of File

IDL_FileEOF()

The IDL_FileEOF() function returns TRUE if the file specified by the Logical Unit Number unit is at EOF, and FALSE otherwise:

```
int IDL_FileEOF(int unit)
```

unit

The Logical Unit Number (LUN) of the file in question.

Flushing Buffered Data

IDL_FileFlushUnit()

File data might be buffered in system memory in order to boost input/output performance. The IDL_FileFlushUnit() function forces any buffered data for the file specified by the Logical Unit Number unit to be written out:

```
int IDL_FileFlushUnit(int unit)
```

unit

The Logical Unit Number (LUN) of the file in question.

Reading a Single Character

IDL_GetKbrd()

The IDL_GetKbrd() function returns the character code of the next available character from IDL_STDIN_UNIT:

```
int IDL_GetKbrd(int should_wait)
```

should_wait

Set this argument to TRUE if IDL_GetKbrd() should wait for a key to be struck, FALSE otherwise.

If should_wait is FALSE and no input characters are waiting in the input stream, IDL_GetKbrd() returns NULL. Otherwise, it waits until a key is struck (if necessary) and then returns its ASCII value. This function will generate an error and return to the interpreter if IDL_STDIN_UNIT is not a terminal.

Output of IDL Variables

IDL_Print() and IDL_PrintF()

The IDL_Print() and IDL_PrintF() functions output the value of IDL_VARIABLEs. IDL_Print() simply outputs to IDL_STDOUT_UNIT, while IDL_PrintF() outputs to a specified unit:

```
void IDL_Print(int argc, IDL_VPTR *argv, char *argk)
void IDL_PrintF(int argc, IDL_VPTR *argv, char *argk)
```

argc

The number of arguments to argv.

argv

IDL_VPTRs of the IDL_VARIABLEs to be output.

argk

Keywords. Set this argument to NULL (`(char *) 0`).

These functions are the implementation of the built-in IDL system procedures PRINT and PRINTF. See "PRINT/PRINTF" in the *IDL Reference Guide* manual for information on the available arguments and the order in which you must specify them.

Adding to the Journal File

IDL_Logit()

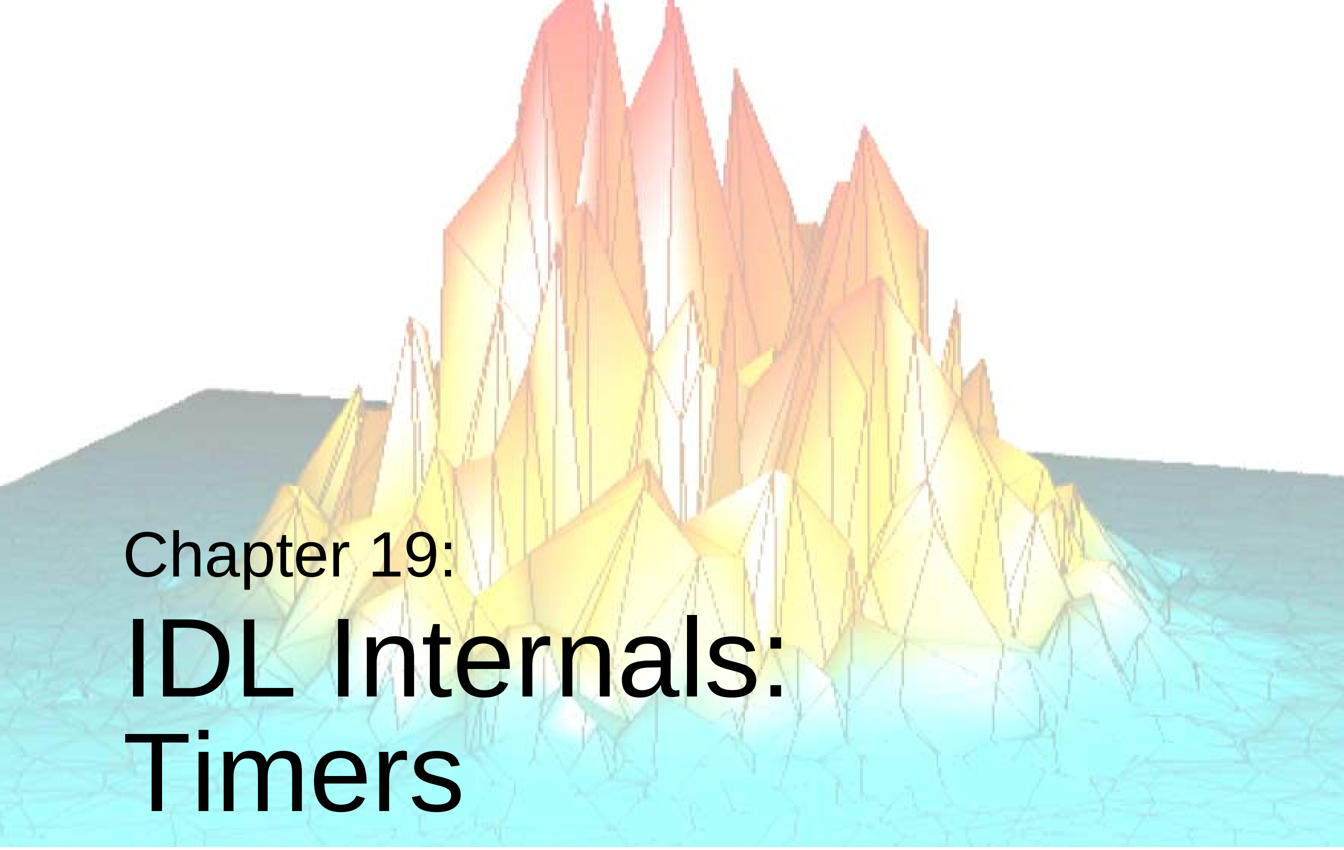
The IDL_Logit() function can be used to add lines of output to the journal log file:

```
void IDL_Logit(char *s)
```

s

A pointer to a NULL terminated string to be added to the journal log file.

If a journal log file is currently open, this function writes the specified string to it on a new line. If no journal file is open, IDL_Logit() returns quietly. The only way to open or close the journal file is via the user-system-level JOURNAL procedure.



Chapter 19: IDL Internals: Timers

This chapter discusses the following topics:

IDL and Timers

The details of how timers work varies widely between operating systems and between variants of the same operating system (different “flavors” of UNIX, for example). IDL’s timer module is intended to provide a stable interface to the rest of IDL, and to isolate the non-portable code in one place.

Under UNIX, IDL’s timer module performs a more important function. UNIX processes contain a single timer that must be shared by the code in the process. When the timer fires, it raises the **SIGALRM** signal which must be caught and handled by the process. The IDL timer routines transparently multiplex this single timer to provide multiple virtual timers.

Under UNIX, IDL provides both blocking and non-blocking timers. Blocking timers put the calling process to sleep until they go off. Non-blocking timers are delivered asynchronously when they fire.

Under Microsoft Windows, only the blocking form of timer requests are supported.

Making Timer Requests

The **IDL_TimerSet()** function registers a timer request. IDL timer requests are one-shot timers. If you wish to have a timer go off repeatedly, your callback function must make a new request each time it is called to set up the next timer.

```
void IDL_TimerSet(length, callback, from_callback, context)
```

where:

length

The length of time to delay before issuing the alarm, in microseconds. You should be aware that other activity on the system, overhead incurred in managing the timers, and non-realtime attributes of the operating system can cause the actual duration of the timer to be longer than requested.

callback

Under UNIX, if **callback** is non-NULL, the timer request is queued and **IDL_TimerSet()** returns immediately. When the alarm is due, the function pointed at by **callback** is called. If **callback** is NULL (and not **from_callback**), the request is queued and **IDL_TimerSet()** blocks until the requested time expires.

Warning

When called, the callback function will be running in signal scope, meaning that it has been called from a signal handler running asynchronously from the rest of the program. There are significant restrictions on what code running in signal scope is allowed to do. Most common C library functions (such as printf()) are disallowed. Consult a book on UNIX programming or your system documentation for details.

Under Windows, **callback** should always be NULL. **IDL_TimerSet()** does not support non-blocking timers on these platforms.

from_callback

Set this argument to TRUE if this invocation is from a callback function previously set up via a call to **IDL_TimerSet()**. Set this argument to FALSE if this is the first invocation. In other words, this argument should only be TRUE if you call **IDL_TimerSet()** from within a timer callback.

context

This argument is a pointer to a variable of type **IDL_TIMER_CONTEXT**, an opaque IDL data type that uniquely identifies a timer request. If this is a top level request (if **from_callback** is FALSE), the context pointed at will be assigned a unique value that identifies the request.

If this request is coming from within a timer callback in order to make another request on the same timer, the context pointed at should contain the value from the previous request.

If **context** is NULL, no context value is returned.

Note

It is an error to queue more than one request using the same callback. The results are undefined.

For the timer module to perform adequately, the time request must be large compared to the run-time of the called function. Re-queuing an extremely short request repeatedly will cause any other requests to starve.

Canceling Asynchronous Timer Requests

Under UNIX, **IDL_TimerCancel()** can be used to cancel a timer request that has not yet been delivered:

```
void IDL_TimerCancel(context)
```

where:

context

A timer request context returned by a previous call to **IDL_TimerSet()**.

Blocking UNIX Timers

Under UNIX operating systems, the delivery of signals such as **SIGALRM** (used to manage timers) can cause system calls to be interrupted. In such cases, the system call returns a status of **-1** and the global **errno** variable is set to the value **EINTR**. It is the caller's responsibility to check for this result and restart the system call when it occurs.

It is easy enough to handle this case when you make system calls directly, but sometimes the problem surfaces in libraries (even those provided by the system, such as `libc`) that are not properly coded against this possibility because the author assumed that no interrupts would occur. There is very little that the end user can do about such libraries except take steps that prevent signals from being raised during these critical sections.

If the IDL timer module is being used to deliver asynchronous events, it is inevitable that the delivery of **SIGALRM** will interfere with this sort of library code. The **IDL_TimerBlock()** function is available under UNIX to suspend the delivery of the timer signal. This can be used to provide a window in which no timer will fire. This routine should always be called in pairs, so the timer doesn't get turned off permanently. It is important to be sure a `longjmp()` (such as caused by calling **IDL_Message()** with the **IDL_MSG_LONGJMP** action code) doesn't happen in the critical region. In addition, this function is not re-entrant.

The effect of blocking timer delivery is that the UNIX **SIGALRM** signal is masked to prevent delivery. If the timer fires during this window of time, the signal will not be delivered until timers are unblocked. At that time, the timer module resumes managing the single real UNIX timer. In the meantime, timer requests are arbitrarily delayed from being queued and processed. Clearly, excessive blocking of the timer can lead to poor timer performance and should only be performed when necessary and on the smallest possible critical section of code. Of course, the act of blocking and unblocking signals requires a context switch into the UNIX kernel and back, making them relatively computationally expensive operations. It is therefore better to block a longer section of code rather than block and unblock around every critical library call.

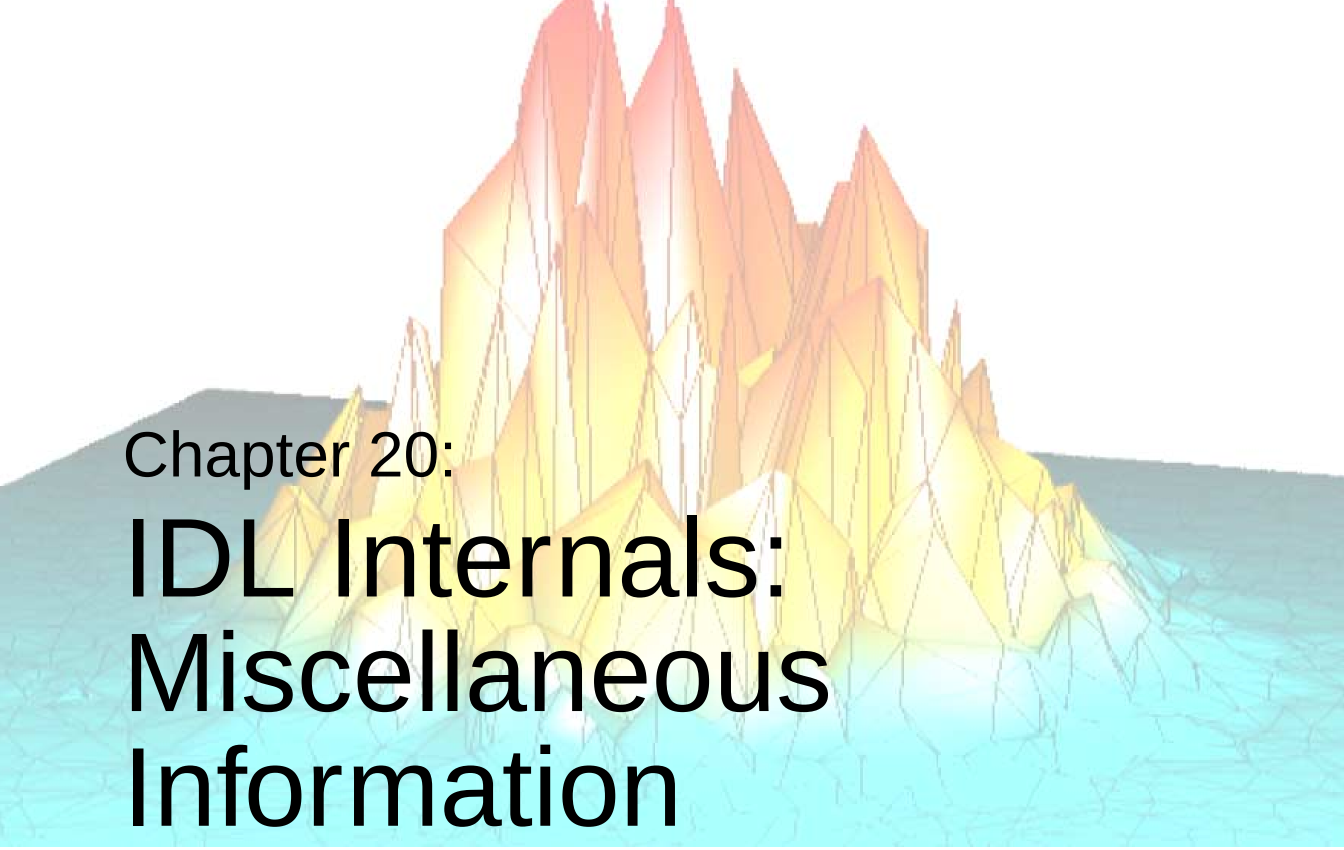
It has been our experience that some UNIX platforms have more problem with this issue than others. You should let experience guide you in deciding when to block signals and when to let them go. Input/Output to device special files under HP-UX and SGI IRIX are known to be especially vulnerable.

```
void IDL_TimerBlock(stop)
```

where:

stop

TRUE if the timer should be suspended, FALSE to restart it.



Chapter 20:

IDL Internals: Miscellaneous Information

This chapter discusses the following topics:

Dynamic Memory

IDL provides access to the dynamic memory allocation routines it uses internally. Use these routines rather than system-provided routines such as **malloc()/free()** when possible.

Warning

The memory pointers returned by the IDL memory allocation routines discussed in this chapter do not necessarily correspond directly to **malloc()**/**free**() calls, or to any other system memory allocation package. You must be careful not to mix memory allocation packages. Memory allocated via a given API can only be freed by the corresponding free call provided by that API. For example, memory allocated by an IDL memory allocation routine can only be freed by the IDL **IDL_MemFree**() function. Memory allocated by **malloc**() can only be freed by **free**().

Failure to follow this rule can lead to memory corruption, including possible crashing of the IDL program.

Please note that code called via CALL_EXTERNAL, or as a system routine (LINKIMAGE, Dynamically Loadable Modules) should not use the IDL dynamic memory routines. Instead, use **IDL_GetScratch()** (see "Getting Dynamic Memory" on page 288) which prevents memory from being lost under error conditions.

Warning

Our experience shows that in situations where **IDL_GetScratch**() is appropriate, use of any other memory allocation mechanism should raise a warning flag to the programmer that something is wrong in their code. Rarely if ever is a direct call to **malloc**()/**free**() reasonable in such a situation — even if it appears to work correctly, you will have to work harder to provide the error handling functionality that **IDL_GetScratch**() provides automatically, or your code will leak memory in such situations.

IDL_MemAlloc()

IDL_MemAlloc() is used to allocate dynamic memory.

```
void *IDL_MemAlloc(IDL_MEMINT n, char *err_str, int action)
```

where:

n

The number of bytes to allocate.

err_str

NULL, or a null terminated text string describing the memory being allocated.

action

An action parameter to be passed to **IDL_Message()** if **IDL_MemAlloc()** is unable to allocate the desired memory and **err_str** is non-NULL.

IDL_MemAlloc() attempts to allocate the desired amount of memory. If the requested amount is allocated, a pointer to the memory is returned. The memory is aligned strictly enough to be suitable for any object.

If the attempt to allocate memory fails and **err_str** is non-NULL, **IDL_Message()** is called as:

```
IDL_Message(IDL_M_CNTGETMEM, action, err_str)
```

If **IDL_Message()** returns, or if **err_str** is NULL and **IDL_Message()** is not called, **IDL_MemAlloc()** returns a NULL pointer indicating its failure.

IDL_MemFree()

Memory allocated via **IDL_MemAlloc()** should only be returned via **IDL_MemFree()**:

```
void IDL_MemFree(REGISTER void *m, char *err_str, int action)
```

m

A pointer to memory previously allocated via **IDL_MemAlloc()**.

err_str

NULL, or a null terminated text string describing the memory being freed.

action

An action parameter to be passed to **IDL_Message()** if unable to free memory and **err_str** is non-NULL.

IDL_MemFree() attempts to free the specified memory. If the attempt to free memory fails and **err_str** is non-NULL, **IDL_Message()** is called as:

```
IDL_Message(IDL_M_CNTFREMEM, action, err_str)
```

The following actions have undefined consequences, and should not be done:

- Returning memory allocated from a source other than **IDL_MemAlloc()**.
- Freeing the same allocation more than once.
- Dereferencing memory once it has been freed.

IDL_MemAllocPerm()

Another memory allocation routine, **IDL_MemAllocPerm()**, exists to allocate dynamic memory that will not be returned for reuse. **IDL_MemAllocPerm()** allocates memory in moderately large units and carves out pieces of these blocks to satisfy its requests. Use of this routine can help minimize the effects of memory fragmentation.

```
void *IDL_MemAllocPerm(IDL_MEMINT n, char *err_str, int action)
```

IDL_MemAllocPerm() takes the same arguments as **IDL_MemAlloc()**, differing only in that the memory allocated will not be freed until the process exits. Do not attempt to free memory allocated by **IDL_MemAllocPerm()**. The results of such an action are undefined.

Exit Handlers

IDL maintains a list of exit handler functions that it calls as part of its shutdown operations. These handlers perform actions such as closing files, wrapping up graphics output, and restoring the user environment to its initial state. Exit handlers accept no arguments and return no value.

A typical declaration would be:

```
void my_exit_handler(void)
{
  /* Cleanup Code Here */
}
```

IDL_ExitRegister()

To register an exit handler, use the **IDL_ExitRegister()** function:

```
void IDL_ExitRegister(IDL_EXIT_HANDLER_FUNC)
```

where IDL_EXIT_HANDLER_FUNC is defined as:

```
typedef void(* IDL_EXIT_HANDLER_FUNC)(void);
```

proc

IDL will call **proc** just before it exits.

The order in which exit handlers are called is undefined, and you should not depend on any particular ordering. If you have several exit handlers and the order in which they are called is important, you should register a single handler that calls all the others in the required order.

Note

Under some operating systems, it is possible that the IDL process will die in an abnormal way that prevents the calling of the exit handlers. For example, under UNIX, receiving some signals (possibly via the **kill(1)** command) will cause the process to die immediately. IDL always calls exit handlers when possible, so this is rarely a significant problem.

User Interrupts

IDL catches certain operating system signals including **SIGINT**, which occurs when the user types the interrupt character (usually Control-C). When the interpreter detects the interrupt character, it sets an internal flag which causes execution of the program to stop at the next sequence statement. The interpreter clears this variable every time it is invoked, and checks to see if it has been set before it executes each statement. This means that when the user presses the interrupt character, the current statement must complete before the interpreter checks the value of the variable and halts execution.

Typical statements do not take long to complete, so this delay is not noticeable. However, some system routines take a long time to complete, and the user can be fooled by the long delay into thinking that IDL is ignoring the interrupt. While the occasional long delay can be annoying, this method of handling interrupts is the only way to maintain acceptable performance in the usual case where no interrupt is pending. Therefore, it is the responsibility of system routines that take a long time to complete to check the value of this internal variable and to clean up and return if **SIGINT** is seen. IDL's Input/Output and FFT routines, among others, do this.

IDL_BailOut()

The **IDL_BailOut()** function is used to sense or set the state of IDL's internal interrupt flag. It returns TRUE if the keyboard interrupt character has been typed, otherwise FALSE.

```
int IDL_BailOut(int stop)
```

where:

stop

Set to FALSE to sense the state of the keyboard interrupt flag without changing its value. Set to TRUE to set the keyboard interrupt flag.

Functions for Returning System Variables

The following functions return the values of certain system variables. Note that these values should be considered READ-ONLY.

IDL_STRING *IDL_SysvVersionArch(void)

This function returns a pointer to the !VERSION.ARCH system variable.

IDL_STRING *IDL_SysvVersionOS(void)

This function returns a pointer to the !VERSION.OS system variable.

IDL_STRING *IDL_SysvVersionOSFamily(void)

This function returns a pointer to the !VERSION.OS_FAMILY system variable.

IDL_STRING *IDL_SysvVersionRelease(void)

This function returns a pointer to the !VERSION.RELEASE system variable.

IDL_STRING *IDL_SysvDirFunc(void)

This function returns a pointer to the !DIR system variable.

IDL_STRING *IDL_SysvErrStringFunc(void)

This function returns a pointer to the !ERROR_STATE.MSG system variable.

IDL_STRING *IDL_SysvSyserrStringFunc(void)

This function returns a pointer to !ERROR_STATE.SYS_MSG system variable.

IDL_LONG IDL_SysvErrorCodeValue(void)

This function returns the value of the !ERROR_STATE system variable.

IDL_LONG IDL_SysvOrderValue(void)

This function returns the value of the !ORDER system variable.

For more information on IDL system variables, see Appendix D, “System Variables” in the *IDL Reference Guide* manual.

Terminal Information

The global variable **IDL_FileTerm** is a structure of type **IDL_TERMINFO**:

```
typedef struct {
  char *name;     /* Name Of Terminal Type */
  char is_tty;    /* True if stdin is a terminal */
  int lines;      /* Lines on screen */
  int columns;    /* Width of output */
} IDL_TERMINFO;
```

Note

Under operating systems that do not support the concept of a terminal (Microsoft Windows) the **name** and **is_tty** fields are not present.

IDL_FileTerm is initialized when IDL is started. Few, if any, user routines will need this information, because user routines should not do their own I/O. User routines that must do their own I/O should use this variable instead of making assumptions about the output device.

Note

Under Microsoft Windows, the **IDL_FileTerm** is not accessible outside of the IDL sharable library, and cannot be directly accessed by user code. Instead, use the functions described in the following section.

Functions for Returning IDL_FileTerm Variable Values

The following functions can be used to return values from the **IDL_FileTerm** variable. They return the same information contained in the global variable, but in a functional form. This is the preferred way to access the **IDL_FileTerm** information, as it will work on any platform.

char *IDL_FileTermName(void)

This function returns the value of **IDL_FileTerm.name**. This function is only available under UNIX.

int IDL_FileTermIsTty(void)

This function returns the value of **IDL_FileTerm.is_tty**. This function is only available under UNIX.

int IDL_FileTermLines(void)

This function returns the value of **IDL_FileTerm.lines**.

int IDL_FileTermColumns(void)

This function returns the value of **IDL_FileTerm.columns**.

Ensuring UNIX TTY State

Under some UNIX operating systems, IDL keeps the users terminal in a *raw mode*, required to implement command line editing. On these platforms, externally linked code that performs output to the terminal will find that the output does not appear as expected. A usual symptom of this is that newline characters ('\n') do not move the cursor to the left margin of the screen, and commands such as more(1) (perhaps started via the C runtime library **system()** function) do not control the screen properly.

This is not an issue for IDL routines such as SPAWN that start sub-programs, because they are written to be aware of this issue and to ensure the TTY is in the correct state before they do their work. Externally linked code can call the **IDL_TTYReset()** function to do the same thing:

```
void IDL_TTYReset(void)
```

This function is available under all operating systems. On systems where such an operation is not needed, it is a stub. On platforms that require the TTY to be managed in this way, this operation ensures that the terminal is returned to its standard configuration.

Type Information

The following read-only global variables provide information about IDL data.

Note

Under Microsoft Windows, these global variables are not available; use the functions listed below to retrieve the values contained in the global variables.

IDL_OutputFormat

An array of pointers to character strings. **IDL_OutputFormat** is indexed by type code, and specifies the default output formats for the different data types (see "Type Codes" on page 258). The default formats are used by the PRINT and STRING built-in routines as well as for type conversions.

IDL_OutputFormatLen

An array of integers. **IDL_OutputFormatLen** gives the length in characters of the corresponding elements of **IDL_OutputFormat**.

IDL_TypeSize

An array of long integers. **IDL_TypeSize** is indexed by type code, and gives the size of the data object used to represent each type.

IDL_TypeName

An array of pointers to character strings. **IDL_TypeName** is indexed by type code, and gives a descriptive string for each type.

Functions for Returning Data Type Variable Values

The following functions can be used to return the values contained in the global variables described above, but in a functional form.

char *IDL_OutputFormatFunc(int type)

Given an IDL type code, this function returns the default output format for that type. This is equivalent to accessing the **IDL_OutputFormat** array.

int IDL_OutputFormatLenFunc(int type)

Given an IDL type code, this function returns the default output format length for that type. This is equivalent to accessing the **IDL_OutputFormatLen** array.

int IDL_TypeSizeFunc(int type)

Given an IDL type code, this function returns the size of the data object used to represent that type. This is equivalent to accessing the **IDL_TypeSize** array.

char *IDL_TypeNameFunc(int type)

Given an IDL type code, this function returns the name of the type as a null terminated character string. This is equivalent to accessing the **IDL_TypeName** array.

User Information

Use the **IDL_GetUserInfo()** function to get information about the current session. This is the sort of information that can be used in the header of files produced by graphics drivers. It is used, for example, by the PostScript driver:

```
void IDL_GetUserInfo(IDL_USER_INFO *user_info)
```

where the **IDL_USER_INFO** struct is defined as:

```
typedef struct {
  char *logname;            /* User’s login name */
  char *homedir;            /* User’s home directory */
  char *pid;                /* The process ID */
  char host[64];            /* Machine name */
  char wd[IDL_MAXPATH+1];   /* Working Directory */
char date[25];              /* Current System Time */
} IDL_USER_INFO;
```

Constants

Preprocessor constants defined in the `idl_export.h` file should be used in preference to hardwired values. To accommodate the needs of various operating systems, some of these constants have different values in different versions of IDL. Those constants that are not discussed elsewhere in this book are listed below.

IDL_TRUE

A more readable alternative to the constant 1.

IDL_FALSE

A more readable alternative to the constant 0.

IDL_REGISTER

Some C compilers are good at allocating registers, and using the C register declaration can cause efficiency to suffer. On the other hand, some C compilers won't put any variables into registers unless register definitions are used. Our solution is to use **IDL_REGISTER** to declare variables we feel should be placed into registers. For machines that we feel have a good register allocation scheme, we define **IDL_REGISTER** to be a null macro. For lesser compilers, it is defined to be the C `register` keyword.

IDL_MAX_ARRAY_DIM

The maximum number of dimensions an array can have.

IDL_MAXIDLEN

The maximum number of characters IDL allows in an identifier (variable names, program names, and so on).

IDL_MAXPATH

The maximum number of characters allowed in a filepath.

Macros

The macros defined in `idl_export.h` handle recurring small jobs. Those macros not discussed elsewhere in this book are covered here.

IDL_ABS(x)

IDL_ABS() accepts a single argument of any numeric C type, and returns its absolute value. **IDL_ABS()** evaluates its argument more than once, so be careful to avoid unwanted side effects, and for efficiency do not call it with a complex expression.

IDL_CARRAY_ELTS(arr)

This macro encapsulates a common C language idiom for determining the number of elements in a statically defined array without requiring the programmer to provide a constant or otherwise hardwire the length. It's use improves the robustness of code that uses it by automatically adapting to any change in the definition of the array without requiring additional programmer effort. This macro corresponds directly to the C expression:

```
sizeof(arr)/sizeof(arr[0])
```

The C compiler evaluates this expression at compile time, so there is no additional runtime cost for using this macro instead of a hardwired constant.

IDL_CHAR(ptr)

IDL_CHAR() casts its argument to be a pointer to **char**. It is used to convert an existing pointer into a generic pointer to a memory location.

IDL_CHARA(addr)

IDL_CHARA() takes the address of its argument and casts it to be a pointer to **char**. It is used to get a generic pointer to a memory location.

IDL_MIN(x,y) and IDL_MAX(x,y)

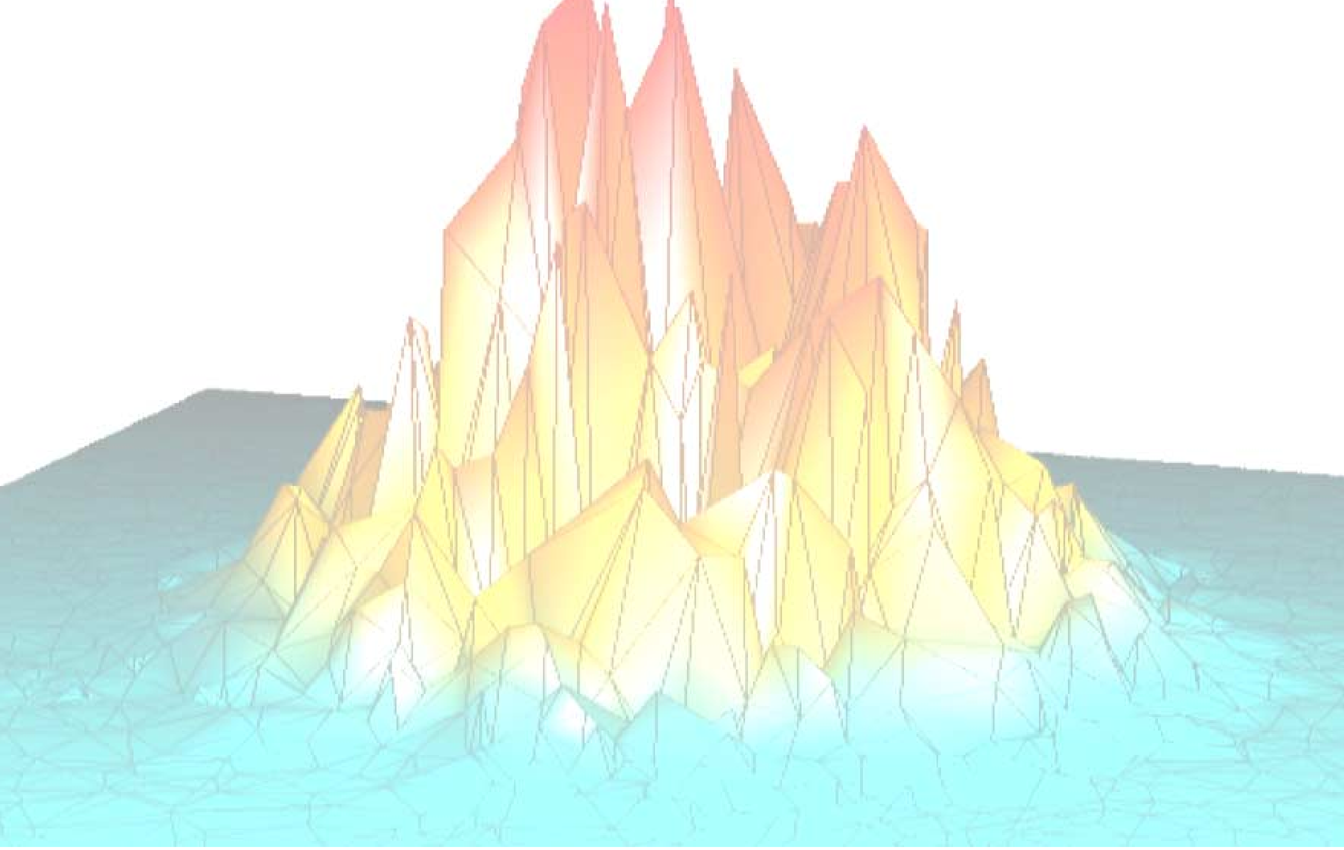
The arguments can be of any numeric C type as long as they are compatible with each other. **IDL_MIN()** and **IDL_MAX()** return the smaller and larger of their two arguments, respectively. These macros evaluate their arguments more than once, so be careful to avoid unwanted side effects, and for efficiency do not call them with a complex expression.

IDL_ROUND_UP(x, m)

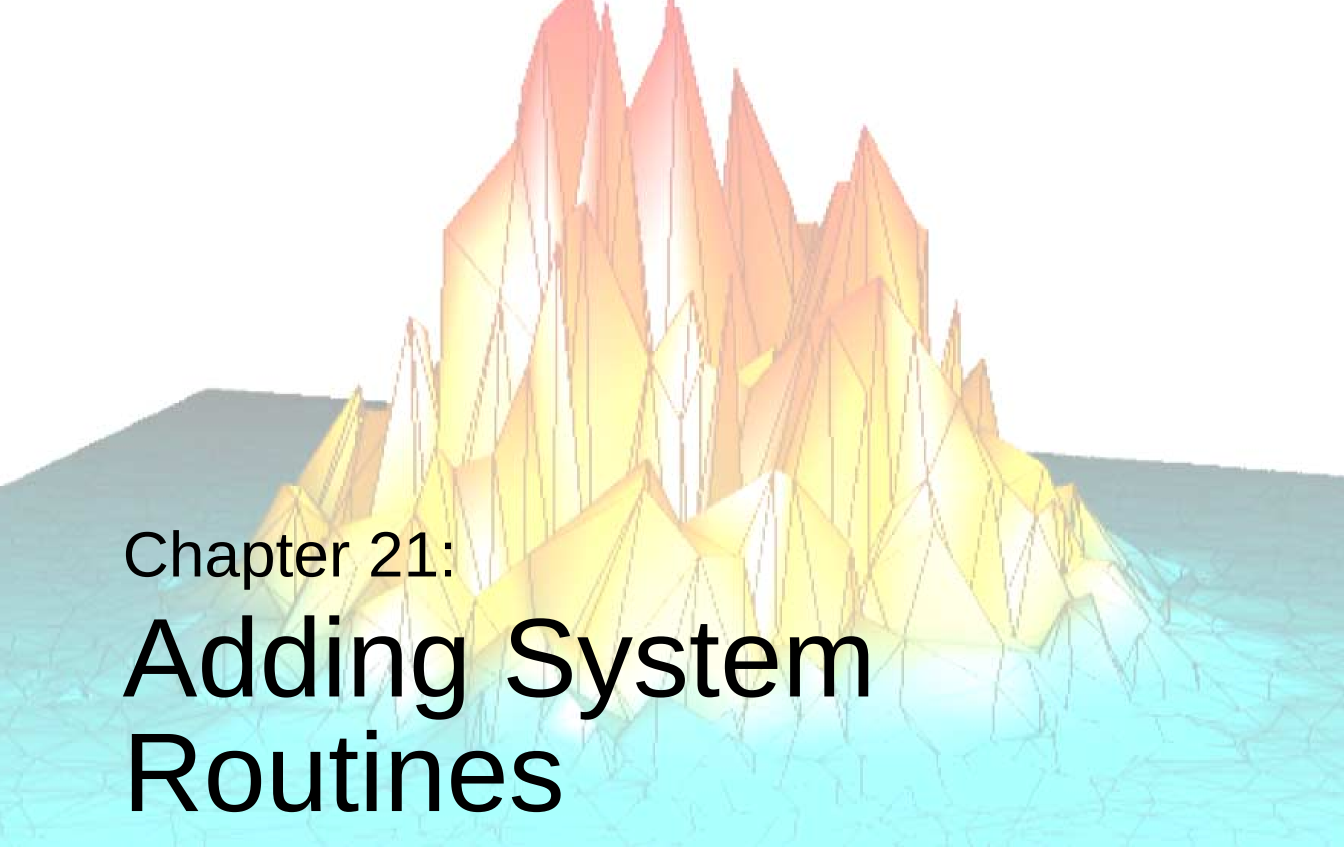
IDL_ROUND_UP() returns the value of **x** rounded up modulo **m**. **m** must be a power of 2. This macro is useful for extending data regions out to a specified alignment.

IDL_TRUE and IDL_FALSE

When performing logical expression evaluation the C programming language, in which IDL is written, treats zero (0) as False, and non-zero as True, and when returning the result of such an expression, uses 1 for True and 0 for False. **IDL_TRUE** is defined as the constant 1, and **IDL_FALSE** is defined as the constant 0. These constants are used internally by IDL.



Part III: Techniques That Use IDL's Internal API



Chapter 21:

Adding System Routines

This chapter discusses the following topics:

IDL and System Routines

An IDL system routine is an IDL procedure or function that is written in a compiled language with an IDL specific interface, and linked into IDL, instead of being written in the IDL language itself.The best way to create an IDL system routine is to compile and link the routine into a sharable library and then to add the routine to IDL at runtime using either the LINKIMAGE procedure or by making your routines part of a Dynamically Loadable Module (DLM).

Note

RSI recommends the use of Dynamically Loadable Modules rather than LINKIMAGE whenever possible. The small additional effort is more than compensated for by the superior integration into IDL.

This chapter explains how to write a system routine, including several examples, and discusses the various options for adding such routines to IDL.

The System Routine Interface

All IDL system routines must supply the same calling interface to the system, differing only in that system functions must return an **IDL_VPTR** to the **IDL_VARIABLE** that contains the result while system procedures do not return anything. Typical system routine definitions are:

```
IDL_VPTR my_function(int argc, IDL_VPTR argv[], char *argk)
void my_procedure(int argc, IDL_VPTR argv[], char *argk)
```

System routines that do not accept keywords are called with two arguments:

argc

The number of elements in **argv**.

argv

An array of **IDL_VPTR**s. These point to the **IDL_VARIABLE**s which comprise the arguments to the function.

System routines that accept keywords are called with an additional third argument:

argk

The keywords which were present when the routine was called. **argk** is an opaque object—the called routine is not intended to understand its contents. **argk** is provided to the function **IDL_KWProcessByOffset()**, which processes the keywords in a standard way. For more information on keywords, see "IDL Internals: Keyword Processing" on page 297.

Example: Hello World

Thanks to the definitive text on the C language (Kernighan and Ritchie, *The C Programming Language*, Prentice Hall, NJ, Second Edition, 1988), the “Hello World” program is often used as an example of a trivial program. Our version of this program is a system function that returns a scalar string containing the text “Hello World!”:

```
#include <stdio.h>
#include "idl_export.h"

IDL_VPTR hello_world(int argc, IDL_VPTR argv[])
{
  return(IDL_StrToSTRING("Hello World!"));
}
```

This is about as simple as an IDL system routine can be. The function **IDL_StrToSTRING()**, returns a temporary variable which contains a scalar string. Since this is exactly what is wanted, **hello_world()** simply returns the variable.

After compiling this function into a sharable object (named, for example, **hello_exe**), we can link it into IDL with the following LINKIMAGE call:

```
LINKIMAGE, 'HELLO_WORLD', 'hello_exe', 1, 'hello_world', $
   MAX_ARGS=0, MIN_ARGS=0
```

We can now issue the IDL command:

```
PRINT, HELLO_WORLD()
```

In response, IDL writes to the screen:

```
Hello World!
```

Example: Doing a Little More (MULT2)

The system function shown in the following figure does a little more than the previous one, though not by much. It expects a single argument, which must be an array. It returns a single-precision, floating-point array that contains the values from the argument multiplied by two.

```
      1 #include <stdio.h>
      2 #include "idl_export.h"
      3
      4 IDL_VPTR mult2(int argc, IDL_VPTR argv[])
      5 {
      6   IDL_VPTR dst, src;
      7   float *src_d, *dst_d;
      8   int n;
      9   src = dst = argv[0];
     10
     11   IDL_ENSURE_SIMPLE(src);
     12   IDL_ENSURE_ARRAY(src);
     13
     14   if (src->type != IDL_TYP_FLOAT)
C    15     src = dst = IDL_CvtFlt(1, argv);
     16
     17   src_d = dst_d = (float *) src->value.arr->data;
     18
     19   if (!(src->flags & IDL_V_TEMP))
     20     dst_d = (float *)
     21       IDL_MakeTempArray(IDL_TYP_FLOAT,src->value.arr->n_dim,
     22                         src->value.arr->dim,
     23                         IDL_ARR_INI_NOP, &dst);
     24
     25   for (n = src->value.arr->n_elts; n--; )
     26      *dst_d++ = 2.0 * *src_d++;
     27
     28   return(dst);
     29 }
```

Table 21-1: mult2.c

Each line is numbered to make discussion easier. These numbers are not part of the actual program. Each line of this routine is discussed below:

1-2

Include the required header files.

4

Every system routine takes the same two or three arguments. **argc** is the number of arguments, **argv** is an array of arguments. This routine does not accept keywords, so **argk** is not present.

6

dst will become a pointer to the resulting variable’s descriptor. **src** points at the input variable which is found in **argv[0]**.

7

src_d and **dst_d** will point to the source and destination data areas.

8

n will contain the number of elements in **src**.

10

Assume, for now, that the input variable will serve as both the source and destination. This will only be true if the parameter is a temporary floating-point array.

11-12

Screen out any input that is not of a basic type, and only allow arrays. A better version of this routine would handle scalar input also, but we want to keep the example simple.

14

If the input is not of **IDL_TYP_FLOAT**, we call the **IDL_CvtFlt()** function to create a floating-point copy of the argument (see “Converting to Specific Types” on page 352 for information about converting types).

Note that the routine could also be written, more efficiently, with a C switch statement which would handle each of the eight possible data types, eliminating conversion of the input parameter. This would be more in the spirit of the IDL language, where system routines work with all possible data types and sizes, but is outside the scope of this example.

17

Here, we initialize the pointers to the source and destination data areas from the array block structure pointed to by the input variable descriptor.

19-23

If the input variable is not a temporary variable, we cannot change its value and return it as the function result. Instead, we allocate a new temporary floating point array into which the result will be placed. Notice how the number of dimensions and their sizes are taken from the source variable array block. See "Array Variables" on page 271 and "Temporary Variables" on page 279.

25

Loop over each element of the arrays.

26

Do the multiplication for each element.

28

Return the temporary variable containing the result.

Testing the Example

Once we have compiled the function and linked it into IDL (possibly using LINKIMAGE), we can use the built-in IDL function INDGEN to test the new function, which we name MULT2. INDGEN returns an array of values with each element set to the value of its array index. Therefore, the statement:

```
PRINT, INDGEN(5)
```

prints the following on the screen:

```
0 1 2 3 4
```

To test our new function we use INDGEN to provide an input argument:

```
PRINT, MULT2(INDGEN(5))
```

The result, as expected, is:

```
0.00000 2.00000 4.00000 6.00000 8.00000
```

Example: A Complete Numerical Routine Example (FZ_ROOTS2)

The following is a complete implementation of the IDL system function FZ_ROOTS, used to find the roots of an *m*-degree complex polynomial, using Laguerre’s method. The result is an *m*-element complex vector. We call this version FZ_ROOTS2 to avoid a name clash with the real routine. FZ_ROOTS2 has an additional keyword, TC_INPUT, that is not present in the real routine.

FZ_ROOTS2 uses the routine **zroots()**, described in section 9.5 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press:

```
void zroots(fcomplex a[], int m, fcomplex roots[], int polish)
```

Quoting from the referenced book:

Given the degree m and the m+1 complex coefficients a[0..m] of the polynomial $\sum_{i=0}^{m} a(i)x^i$, this routine successively calls `laguer` and finds all m complex roots in roots[1..m]. The boolean variable `polish` should be input as true (1) if polishing (also by Laguerre’s method) is desired, false (0) if the roots will be subsequently polished by other means.

FZ_ROOTS2 will support both single and double precision complex values as well as give the caller control over the error tolerance, which is hard wired into the Numerical Recipes code as a C preprocessor constant named EPS. In order to support these requirements, we have copied the **zroots()** function given in the book and altered it to support both data types and make EPS a user specified parameter, giving two functions:

```
void zroots_f(fcomplex a[], int m, fcomplex roots[], int polish,
              float eps);

void zroots_d(dcomplex a[], int m, dcomplex roots[], int polish,
              double eps);
```

Note that **fcomplex** and **dcomplex** are Numerical Recipes defined types that happen to have the same definition as the IDL types **IDL_COMPLEX** and **IDL_DCOMPLEX**, a convenient fact that eliminates some type conversion issues.

The definition of FZ_ROOTS2 from the IDL user perspective is:

Calling Sequence

Result = FZ_ROOTS2(C)

Arguments

C

A vector of length $m+1$ containing the coefficients of the polynomial, in ascending order.

Keywords

DOUBLE

FZ_ROOTS2 normally uses the type of C to determine the type of the computation. If DOUBLE is specified, it overrides this default. Setting DOUBLE to a non-zero value causes the computation type and the result to be double precision complex. Setting it to zero forces single precision complex.

EPS

The desired fractional accuracy. The default value is 2.0×10^{-6}.

NO_POLISH

Set this keyword to suppress the usual polishing of the roots by Laguerre’s method.

TC_INPUT

If present, TC_INPUT specifies a named variable that will be assigned the input value C, with its type converted to the type of the result.

Example

The following figure gives the code for fzroots2.c,. This is ANSI C code that implements FZ_ROOTS2. The line numbers are not part of the code and are present to make the discussion easier to follow. Each line of this routine is discussed below.

C

```
 1 #include <stdio.h>
 2 #include <stdarg.h>
 3 #include "idl_export.h"
 4 #include <nr/nr.h>
 5
 6 IDL_VPTR fzroots2(int argc, IDL_VPTR *argv, char *argk)
 7 {
 8   typedef struct {
 9     IDL_KW_RESULT_FIRST_FIELD; /* Must be first entry in this structure */
10     int force_type;
11     IDL_LONG do_double;
12     double eps;
13     IDL_LONG no_polish;
14     IDL_VPTR tc_input;
15   } KW_RESULT;
16   static IDL_KW_PAR kw_pars[] = {
17     {"DOUBLE", IDL_TYP_LONG, 1, 0,
18      IDL_KW_OFFSETOF(force_type), IDL_KW_OFFSETOF(do_double) },
19     { "EPS", IDL_TYP_DOUBLE, 1, 0, 0, IDL_KW_OFFSETOF(eps) },
20     { "NO_POLISH", IDL_TYP_LONG, 1, IDL_KW_ZERO,
21       0, IDL_KW_OFFSETOF(no_polish) },
22     { "TC_INPUT", 0, 1, IDL_KW_OUT|IDL_KW_ZERO,
23       0, IDL_KW_OFFSETOF(tc_input) },
24     { NULL }
25   };
26
27   KW_RESULT kw;
28   IDL_VPTR result;
29   IDL_VPTR c_raw;
30   IDL_VPTR c_tc;
31   IDL_MEMINT m;
32   void *outdata;
33   IDL_ARRAY_DIM dim;
34   int rtype;
35   static IDL_ALLTYPES zero;
36
37   kw.eps = 2.0e-6;
38   (void) IDL_KWProcessByOffset(argc, argv, argk, kw_pars,&c_raw,1,&kw);
39
40   IDL_ENSURE_ARRAY(c_raw);
41   IDL_ENSURE_SIMPLE(c_raw);
42   if (c_raw->value.arr->n_dim != 1)
43   IDL_Message(IDL_M_NAMED_GENERIC, IDL_MSG_LONGJMP,
44               "Input argument must be a column vector.");
45   m = c_raw->value.arr->dim[0];
46   if (--m <= 0)
47     IDL_Message(IDL_M_NAMED_GENERIC, IDL_MSG_LONGJMP,
48                 "Input array does not have enough elements");
```

Table 21-2: fzroots2.c

C

```
49  if (kw.tc_input)
50     IDL_StoreScalar(kw.tc_input, IDL_TYP_LONG, &zero);
51
52   if (kw.force_type) {
53     rtype = kw.do_double ? IDL_TYP_DCOMPLEX : IDL_TYP_COMPLEX;
54   } else {
55     rtype = ((c_raw->type == IDL_TYP_DOUBLE)
56              || (c_raw->type == IDL_TYP_DCOMPLEX))
57       ? IDL_TYP_DCOMPLEX : IDL_TYP_COMPLEX;
58   }
59   dim[0] = m;
60   outdata = (void *)
61     IDL_MakeTempArray(rtype,1,dim,IDL_ARR_INI_NOP,&result);
62
63   if (c_raw->type == rtype) {
64     c_tc = c_raw;
65   } else {
66     c_tc = IDL_BasicTypeConversion(1, &c_raw, rtype);
67   }
68
69   if (rtype == IDL_TYP_COMPLEX) {
70     zroots_f((fcomplex *) c_tc->value.arr->data, m,
71              ((fcomplex *)outdata)-1,!kw.no_polish,(float) kw.eps);
72   } else {
73     zroots_d((dcomplex *) c_tc->value.arr->data, m,
74              ((dcomplex *) outdata) - 1, !kw.no_polish, kw.eps);
75   }
76
77   if (kw.tc_input) IDL_VarCopy(c_tc, kw.tc_input);
78   else if (c_raw != c_tc) IDL_Deltmp(c_tc);
79
80   IDL_KW_FREE;
81   return result;
82 }
```

Table 21-2: (Continued) fzroots2.c

4

`nr.h` is the header file provided with Numerical Recipes in C code.

6

FZROOTS2 has the usual three standard arguments.

10

kw.force_type will be TRUE if the user specifies the DOUBLE keyword. In this case, the value of the DOUBLE keyword will determine the result type without regard for the type of the input argument.

If the user specifies DOUBLE, a zero value forces a single precision complex result and non-zero forces double precision complex.

12

The value of the EPS keyword.

13

The value of the NO_POLISH keyword.

14

The value of the TC_INPUT keyword.

16

This array defines the keywords accepted by FZ_ROOTS2.

17

Since setting DOUBLE to 0 has a different meaning than not specifying the keyword at all, **kw.force_type** is used to detect the fact that the keyword is set independent of its value.

19

The EPS keyword allows the user to specify the **kw.eps** tolerance parameter. **kw.eps** is specified as double precision to avoid losing accuracy for double precision computations—it will be converted to single precision if necessary. The default value for this keyword is non-zero, so no zeroing is specified here. If the user includes the EPS keyword, the value will be placed in **kw.eps**, otherwise **kw.eps** will not be changed.

20

This keyword lets the user suppress the usual polishing performed by **zroots()**. Since specifying a value of 0 is equivalent to not specifying the keyword at all, **IDL_KW_ZERO** is used to initialize the variable.

22

If present, TC_INPUT is an output keyword that will have the type converted value of the input argument stored in it. By specifying **IDL_KW_OUT** and **IDL_KW_ZERO**, we ensure that TC_INPUT is either zero or a pointer to a valid IDL variable.

27

The results of keyword processing will all be written to this variable by **IDL_KWProcessByOffset()**.

28

This variable will receive the function result.

29

The input argument prior to any type conversion.

30

The type converted input variable. If the input variable is already of the correct type, this will be the same as **c_raw**, otherwise it will be different.

31

The value of *m* to be passed to **zroots()**.

32

Pointer to the data area of the result variable. We declare it as `(void *)` so that it can point to data of any type.

33

Used to specify dimensions of the result. This will always be a vector of *m* elements.

34

IDL type code for result variable.

35

Used by **IDL_StoreScalar()** to type check the TC_INPUT keyword. It is declared as static to ensure it is initialized to zero.

37

Set the default EPS value before doing keyword processing. If the user specifies EPS, the supplied value will override this. Otherwise, this value will still be in **kw.eps** and will be passed to **zroots()** unaltered.

38

Perform keyword processing.

40-41

Ensure that the input argument is an array, and is one of the basic types (not a file variable or structure).

42-44

The input variable must be a vector, and therefore should have only a single dimension.

45-48

Ensure that the input variable is long enough for *m* to be non-zero. *m* is one less than the number of elements in the input vector, so this is equivalent to saying that the input must have at least 2 elements.

49

If the TC_INPUT keyword was present, use **IDL_StoreScalar()** to make sure the named variable specified can receive the converted input value. A nice side effect of this operation is that any dynamic memory currently being used by this variable will be freed now instead of later after we have allocated other dynamic memory. This freed memory might be immediately reusable if it is large enough, which would reduce memory fragmentation and lower overall memory requirements.

52

If the user specified the DOUBLE keyword, it is used to control the resulting type, otherwise the input argument type is used to decide.

53

The DOUBLE keyword was specified. If it is non-zero, use **IDL_TYP_DCOMPLEX**, otherwise **IDL_TYP_COMPLEX**.

55-57

Use the input type to decide the result type. If the input is **IDL_TYP_DOUBLE** or **IDL_TYP_DCOMPLEX**, use **IDL_TYP_DCOMPLEX**, otherwise **IDL_TYP_COMPLEX**.

59-61

Create the output variable that will be passed back as the result of FZ_ROOTS2.

63-67

If necessary, convert the input argument to the result type. This is done *after* creation of the output variable because it is likely to have a short lifetime. If it does get freed at the end of this routine, it won't cause memory fragmentation by leaving a hole in the process virtual memory.

69

The version of **zroots()** to call depends on the data type of the result.

70-71

Single precision complex. Note that the outdata pointer is decremented by one element. This compensates for the fact that the Numerical Recipe routine will index it from [1..m] rather than [0..m-1] as is the usual C convention. Also, **kw.eps** is cast to single precision.

73-74

Double precision complex case.

77

If the user specified the TC_INPUT keyword, copy the type converted input into the keyword variable. Since **VarCopy()** frees its source variable if it is a temporary variable, we are relieved of the usual responsibility to call **IDL_Deltmp()** if **c_tc** contains a temporary variable created on line 66.

78

The user didn't specify the TC_INPUT keyword. In this case, if we allocated **c_tc** on line 66, we must free it before returning.

80

Free any resources allocated by keyword processing.

81

Return the result.

Example: An Example Using Routine Design Iteration (RSUM)

We now show how a simple routine can be developed in stages. RSUM is a function that returns the running sum of the values in its single input argument. We will present three versions of this routine, each one of which represents an improvement in functionality and flexibility.

All three versions use the function **IDL_MakeTempFromTemplate(),** described in "Creating A Temporary Variable Using Another Variable As A Template" on page 283. The result of RSUM always has the same general shape and dimensions as the input argument. **IDL_MakeTempFromTemplate()** encapsulates the task of creating a temporary variable of the desired type and shape using the input argument as a template.

Running Sum (Example 1)

The first example of RSUM is very simple. Here is a simple “Reference Manual” style description of it:

RSUM1

Compute a running sum on the array input. The result is a floating point array of the same dimensions.

Calling Sequence

Result = RSUM1(Array)

Arguments

Array

Array for which a running sum will be computed.

This is a minimal design that lacks some important characteristics that IDL system routines usually embody:

- It does not handle scalar input.
- The type of the input is inflexible. IDL routines usually try to handle any input type and do whatever type conversions are necessary.
- The result type is always single precision floating point. IDL routines usually perform computations in the type of the input arguments and return a value of the same type.

We will improve on this design in our subsequent attempts. The code to implement RSUM1 is shown in the following figure. The line numbers are not part of the code and are present to make the discussion easier to follow. Each line of this routine is discussed below:

C

```
 1 IDL_VPTR IDL_rsum1(int argc, IDL_VPTR argv[])
 2 {
 3   IDL_VPTR v;
 4   IDL_VPTR r;
 5   float *f_src;
 6   float *f_dst;
 7   IDL_MEMINT n;
 8
 9
10   v = argv[0];
11   if (v->type != IDL_TYP_FLOAT)
12     IDL_Message(IDL_M_NAMED_GENERIC, IDL_MSG_LONGJMP,
13                 "argument must be float");
14   IDL_ENSURE_ARRAY(v);
15   IDL_EXCLUDE_FILE(v);
16
17   f_dst = (float *)
18     IDL_VarMakeTempFromTemplate(v, IDL_TYP_FLOAT,
19                                 (IDL_StructDefPtr) 0, &r, FALSE);
20   f_src = (float *) v->value.arr->data;
21   n = v->value.arr->n_elts - 1;
22   *f_dst++ = *f_src++;/* First element */
23   for (; n--; f_dst++) *f_dst = *(f_dst - 1) + *f_src++;
24
25   return r;
26 }
```

Table 21-3: Code for IDL_rsum1()

1

The standard signature for an IDL system function that does not accept keywords.

3

This variable is used to access the input argument in a convenient way.

4

This **IDL_VPTR** will be used to return the result.

5–6

As the running sum is computed, **f_src** will point at the input data and **f_dst** will point at the output data.

7

The number of elements in the input.

10

Obtain the input variable pointer from argv[0].

11

If the input is not single precision floating point, throw an error and quit. This is overly rigid. Real IDL routines would attempt to either type convert the input or do the computation in the input type.

14

This version can only handle arrays. If the user passes a scalar, it throws an error.

15

This routine cannot handle ASSOC file variables. Most IDL routines exclude such variables as they do not contain any data to work with. ASSOC variable data usually comes into a routine as the result of an expression that yields a temporary variable (e.g. `TMP = RSUM(MY_ASSOC_VAR(2))`).

17

Create a single precision floating point temporary variable of the same size as the input variable and get a pointer to its data area.

20

Get a pointer to the data area of the input variable. At this point we know this variable is always a floating point array.

21

The number of data elements in the input.

22-23

The running sum computation.

25

Return the result.

Running Sum (Example 2)

In our second example of RSUM, we improve on version 1 in several ways:

- RSUM2 accepts scalar input.
- If the input is not of floating type, we type convert it instead of throwing an error.
- If the input is a temporary variable of the correct type, we will do the running sum computation in place and return the input as our result variable rather than creating an extra temporary. This optimization reduces memory use, and can have positive effects on dynamic memory fragmentation.

As always, the first step in writing a system routine is to write a simple description of its interface and intended behavior:

RSUM2

Compute a running sum on the input. The result is a floating point variable with the same structure.

Calling Sequence

Result = RSUM2(Input)

Arguments

Input

Scalar or array data of any numeric type for which a running sum will be computed.

The following is the code for RSUM2:

C

```c
 1 IDL_VPTR IDL_rsum2(int argc, IDL_VPTR argv[])
 2 {
 3   IDL_VPTR v;
 4   IDL_VPTR r;
 5   float *f_src;
 6   float *f_dst;
 7   IDL_MEMINT n;
 8
 9
10   v = IDL_BasicTypeConversion(1, argv, IDL_TYP_FLOAT);
11   /* IDL_BasicTypeConversion calls IDL_ENSURE_SIMPLE, so
12      skip it here. */
13   IDL_VarGetData(v, &n, (char **) &f_src, FALSE);
14
15   /* Get a result var, reusing the input if possible */
16   if (v->flags & V_TEMP) {
17     r = v;
18     f_dst = f_src;
19   } else {
20     f_dst = (float *)
21       IDL_VarMakeTempFromTemplate(v, IDL_TYP_FLOAT,
22                                    (IDL_StructDefPtr) 0, &r, FALSE);
23   }
24
25   *f_dst++ = *f_src++;/* First element */
26   n--;
27   for (; n--; f_dst++) *f_dst = *(f_dst - 1) + *f_src++;
28
29   return r;
30 }
```

Table 21-4: Code for IDL_rsum2().

Discussion of the code for the improvements introduced in this version follow:

10

If the input has the wrong type, obtain one of the right type. If it was already of the correct type, **IDL_BasicTypeConversion()** will simply return the input value without allocating a temporary variable. Hence, no explicit check for that is required. Also, if the input argument cannot be converted to the desired type (e.g. it is a structure or file variable) **IDL_BasicTypeConversion()** will throw an error. Hence, we know that the result from this function will be what we want without requiring any further checking.

13

IDL_VarGetData() is a more elegant way to obtain a pointer to variable data along with a count of elements. A further benefit is that it automatically handles scalar variables which removes the restriction from RSUM1.

15–23

If the input variable is a temporary, we will do the computation in place and return the input. Otherwise, we create a temporary variable of the desired type to be the result.

Note that if **IDL_BasicTypeConversion()** returned a pointer to anything other than the passed in value of **argv[0]**, that value will be a temporary variable which will then be turned into the function result by this code. Hence, we never free the value from **IDL_BasicTypeConversion()**.

Running Sum (Example 3)

RSUM2 is a big improvement over RSUM1, but it still suffers from the fact that all computation is done in a single data type. A real IDL system routine always tries to perform computations in the most significant type presented by its arguments. In a single argument case like RSUM, that would mean doing computations in the input data type whatever that might be. Our final version, RSUM3, resolves this shortcoming.

RSUM3

Compute a running sum on the input. The result is a variable with the same type and structure as the input.

Calling Sequence

Result = RSUM3(Input)

Arguments

Input

Scalar or array data of any numeric type for which a running sum will be computed.

The code for **RSUM3** is given in the following figure. Discussion of the code for the improvements introduced in this version follow:

```
cx_public IDL_VPTR IDL_rsum3(int argc, IDL_VPTR argv[])
{
   IDL_VPTR v, r;
   union {
     char *sc;                 /* Standard char */
     UCHAR *c;                 /* IDL_TYP_BYTE */
     IDL_INT *i;               /* IDL_TYP_INT */
     IDL_UINT *ui;             /* IDL_TYP_UINT */
     IDL_LONG *l;              /* IDL_TYP_LONG */
     IDL_ULONG *ul;            /* IDL_TYP_ULONG */
     IDL_LONG64 *l64;          /* IDL_TYP_LONG64 */
     IDL_ULONG64 *ul64;        /* IDL_TYP_ULONG64 */
     float *f;                 /* IDL_TYP_FLOAT */
     double *d;                /* IDL_TYP_DOUBLE */
     IDL_COMPLEX *cmp;         /* IDL_TYP_COMPLEX */
     IDL_DCOMPLEX *dcmp;       /* IDL_TYP_DCOMPLEX */
} src, dst;
IDL_LONG n;


v = argv[0];
if (v->type == IDL_TYP_STRING)
  v = IDL_BasicTypeConversion(1, argv, IDL_TYP_FLOAT);
IDL_VarGetData(v, &n, &(src.sc), TRUE);
n--;                           /* First is a special case */   C

/* Get a result var, reusing the input if possible */
if (v->flags & IDL_V_TEMP) {
  r = v;
  dst = src;
} else {
  dst.sc = IDL_VarMakeTempFromTemplate(v, v->type,
                                       (IDL_StructDefPre) 0, &r, FALSE);
}

#define DOCASE(type, field) \
case type: for (*dst.field++ = *src.field++; n--;dst.field++)\
          *dst.field = *(dst.field - 1) + *src.field++; break

#define DOCASE_CMP(type, field) case type: \
for (*dst.field++ = *src.field++; n--; \
     dst.field++, src.field++) { \
  dst.field->r = (dst.field - 1)->r + src.field->r; \
  dst.field->i = (dst.field - 1)->i + src.field->i; } \
break

   switch (v->type) {
   DOCASE(IDL_TYP_BYTE, c);
   DOCASE(IDL_TYP_INT, i);
```

Table 21-5: Code for IDL_rsum3.

```
  49   DOCASE(IDL_TYP_LONG, l);
  50   DOCASE(IDL_TYP_FLOAT, f);
  51   DOCASE(IDL_TYP_DOUBLE, d);
  52   DOCASE_CMP(IDL_TYP_COMPLEX, cmp);
  53   DOCASE_CMP(IDL_TYP_DCOMPLEX, dcmp);
  54   DOCASE(IDL_TYP_UINT, ui);
  55   DOCASE(IDL_TYP_ULONG, ul);
  56   DOCASE(IDL_TYP_LONG64, l64);
C 57   DOCASE(IDL_TYP_ULONG64, ul64);
  58   default: IDL_Message(IDL_M_NAMED_GENERIC, IDL_MSG_LONGJMP,
  59                        "unexpected type");
  60   }
  61 #undef DOCASE
  62 #undef DOCASE_CMP
  63
  64 return r;
  65 }
```

Table 21-5: (Continued) Code for IDL_rsum3.

17

f_src and **f_dst** are no longer pointers to float. They are now the **IDL_ALLPTR** type, which can point to data of any IDL type. To reflect this change in scope, the leading **f_** prefix has been dropped.

22-23

Strings are the only input type that now require conversion. The other types can either support the computation, or are not convertable to a type that can.

36-38

The code for the running sum computation is logically the same for all non-complex data types, differing only in the **IDL_ALLPTR** field that is used for each type.

Using a macro for this means that the expression is only typed in once, and the C compiler automatically fills in the different parts for each data type. This is less error prone than entering the expression manually for each type, and leads to more readable code. This is one of the rare cases where a macro makes things *more* reliable and readable.

39-44

A macro for the 2 complex types.

46-60

A switch statement that uses the macros defined above to perform the running sum on all possible types. Note the default case, which traps attempts to compute a running sum on structures.

61-62

Don’t allow the macros used in the above switch statement to remain defined beyond the scope of this function.

Registering Routines

The **IDL_SysRtnAdd()** function adds system routines to IDL's internal tables of system functions and procedures. As a programmer, you will need to call this function directly if you are linking a version of IDL to which you are adding routines, although this is very rare and not considered to be a good practice for maintainability reasons. More commonly, you use **IDL_SysRtnAdd()** in the **IDL_Load()** function of a Dynamically Loadable Module (DLM). DLMs are discussed in "Dynamically Loadable Modules" on page 450.

Note

LINKIMAGE or DLMs are the preferred way to add system routines to IDL because they do not require building a separate IDL program. Of the two, RSI recommends the use of DLMs whenever possible. These mechanisms are discussed in the following sections of this chapter.

Syntax

```
int IDL_SysRtnAdd(IDL_SYSFUN_DEF2 *defs, int is_function, int cnt)
```

It returns True if it succeeds in adding the routine or False in the event of an error.

Arguments

defs

An array of **IDL_SYSFUN_DEF2** structures, one per routine to be declared. This array must be defined with the C language **static** storage class because IDL keeps pointers to it. **defs** must be sorted by routine name in ascending lexical order.

is_function

Set this parameter to IDL_TRUE if the routines in **defs** are functions, and IDL_FALSE if they are procedures.

cnt

The number of **IDL_SYSFUN_DEF2** structures contained in the **defs** array.

The definition of **IDL_SYSFUN_DEF2** is:

```
typedef IDL_VARIABLE *(* IDL_SYSTRN_GENERIC)();

typedef struct {
  IDL_SYSRTN_GENERIC funct_addr;
  char *name;
  unsigned short arg_min;
  unsigned short arg_max;
  int flags
  void *extra;
} IDL_SYSFUN_DEF2;
```

IDL_VARIABLE structures are described in "The IDL_VARIABLE Structure" on page 267.

funct_addr

Address of the function implementing the system routine.

name

The name by which the routine is to be invoked from within IDL. This should be a pointer to a null terminated string. The name should be capitalized. If the routine is an object method, the name should be fully qualified, which means that it should include the class name at the beginning followed by two consecutive colons, followed by the method name (e.g. `CLASS::METHOD`).

arg_min

The minimum number of arguments allowed for the routine.

arg_max

The maximum number of arguments allowed for the routine. If the routine does not place an upper value on the number of arguments, use the value **IDL_MAXPARAMS**.

flags

A bitmask that provides additional information about the routine. Its value can be any combination of the following values (bitwise OR-ed together to specify more than one at a time) or zero if no options are necessary:

IDL_SYSFUN_DEF_F_OBSOLETE

IDL should issue a warning message if this routine is called and !WARN.OBS_ROUTINE is set.

IDL_SYSFUN_DEF_F_KEYWORDS

This routine accepts keywords as well as plain arguments.

IDL_SYSFUN_DEF_F_METHOD

This routine is an object method.

extra

Reserved to Research Systems, Inc. The caller should set this to 0.

Example

The following example shows how to register a system routine linked directly with IDL. For simplicity, everything is placed in a single file. Normally, you would modularize things to allow easier code maintenance.

```
#include <stdio.h>
#include "idl_export.h"

void prox1(int argc, IDL_VPTR argv[])
{
  printf("prox1 %d\n", IDL_LongScalar(argv[0]));
}

main(int argc, char *argv[])
{
  static IDL_SYSFUN_DEF2 new_pros[] = {
    {(IDL_SYSRTN_GENERIC) prox1, "PROX1", 1, 1, 0, 0}
  };

  if (!IDL_SysRtnAdd(new_pros, IDL_FALSE, 1))
    IDL_Message(IDL_M_GENERIC, IDL_MSG_RET,
     "Error adding system routine");
  return IDL_Main(0, argc, argv);
}
```

This adds a system procedure named **PROX1** which accepts a single argument. It converts this argument to a scalar longword integer and prints it.

Enabling and Disabling System Routines

The following IDL internal functions allow the enabling and/or disabling of IDL system routines. Disabled routines throw an error when called from IDL code instead of performing their usual functions.

These routines are primarily of interest to authors of Runtime or Callable IDL applications.

Enabling Routines

The **IDL_SysRtnEnable()** function is used to enable and/or disable system routines.

Syntax

```
void IDL_SysRtnEnable(int is_function, IDL_STRING *names,
                      IDL_MEMINT n, int option,
                      IDL_SYSRTN_GENERIC disfcn)
```

Arguments

is_function

Set to TRUE if functions are being manipulated, FALSE for procedures.

names

NULL, or an array of names of routines.

n

The number of names in **names**.

option

One of the values from the following table which specify what this routine should do.

Bit	Description
IDL_SRE_ENABLE	Enable specified routines.
IDL_SRE_ENABLE_EXCLUSIVE	Enable specified routines and disable all others.
IDL_SRE_DISABLE	Disable specified routines.
IDL_SRE_DISABLE_EXCLUSIVE	Disable specified routines and enable all others.

*Table 21-6: Values for **option** Argument*

disfcn

NULL, or address of an IDL system routine to be called by the IDL interpreter for these disabled routines. If this argument is not provided, a default routine is used.

Result

All routines are enabled/disabled as specified. If a non-existent routine is specified, it is quietly ignored. Attempts to enable routines disabled for licensing reasons are also quietly ignored.

Note

The routines CALL_FUNCTION, CALL_METHOD (function and procedure), CALL_PROCEDURE, and EXECUTE are not real system routines, but are actually special cases that result in different IDL pcode. For this reason, they cannot be disabled. However, anything they *can* call can be disabled, so this is not a serious drawback.

Obtaining Enabled/Disabled Routine Names

The **IDL_SysRtnGetEnabledNames()** function can be used to obtain the names of all system routines which are currently enabled or disabled, either due to licensing reasons (i.e., some routines are disabled in IDL demo mode) or due to a call to **IDL_SysRtnEnable()**.

Syntax

```
void IDL_SysRtnGetEnabledNames(int is_function,
                               IDL_STRING *str, int
enabled)
```

Arguments

is_function

Set to TRUE if a list of functions is desired, FALSE for a list of procedures.

str

Points to a buffer of IDL_STRING descriptors to fill in. The caller must call **IDL_SysRtnNumEnabled()** to determine how many such routines exist, and this buffer must be large enough to hold that number.

enabled

Set to TRUE to receive names of enabled routines, FALSE to receive names of disabled ones.

Result

The memory supplied via str is filled in with the desired names.

Obtaining the Number of Enabled/Disabled Routines

The **IDL_SysRtnGetEnabledNames()** function requires you to supply a buffer large enough to hold all of the names to be returned. **IDL_SysRtnNumEnabled()** can be called to obtain the number of such routines, allowing you to properly size the buffer.

Syntax

```
IDL_MEMINT IDL_SysRtnNumEnabled(int is_function, int enabled)
```

Arguments

is_function

Set to TRUE if the number of functions is desired, FALSE for procedures.

enabled

Set to TRUE to receive number of enabled routines, FALSE to receive number of disabled ones.

Result

Returns the requested count.

Obtaining the Real Function Pointer

The **IDL_SysRtnGetRealPtr()** routine returns the pointer to the actual internal IDL function that implements the system function or procedure of the specified name.

This routine can be used to interpose your own code in between IDL and the actual routine. This process is sometimes called *hooking* in other systems. To implement such a hook function, you must use the **IDL_SysRtnEnable()** function to register the interposed routine, which in turn uses **IDL_SysRtnGetRealPtr()** to obtain the actual IDLfunction pointer for the routine.

Syntax

```
IDL_SYSRTN_GENERIC IDL_SysRtnGetRealPtr(int is_function,
                                        char *name)
```

Arguments

is_function

Set to TRUE if functions are being manipulated, FALSE for procedures.

name

The name of function or procedure for which the real function pointer is required.

Result

If the specified routine...

- exists and is not disabled, it’s function pointer is returned.
- does not exist, a NULL pointer is returned.
- has been disabled by the user, its actual function pointer is returned.
- has been disabled for licensing reasons, the real function pointer does not exist, and the pointer to its stub is returned.

Note

This routine can cause an IDL_MSG_LONGJMP message to be issued if the function comes from a DLM and the DLM load fails due to memory allocation errors. Therefore, it must not be called unless the IDL interpreter is active. The prime intent for this routine is to call it from the stub routine of a disabled function when the interpreter invokes the associated system routine.

Obtaining the IDL Name of the Current System Routine

To get the IDL name for the currently executing system routine, use the **IDL_SysRtnGetCurrentName()**.

Syntax

```
char *IDL_SysRtnGetCurrentName(void)
```

This function returns a pointer to the name of the currently executing system routine. If there is no currently executing system routine, a NULL (0) pointer is returned.

This routine will never return NULL if called from within a system routine.

LINKIMAGE

The IDL user level LINKIMAGE procedure makes the functionality of the **IDL_SysRtnAdd**() function available to IDL programs. It allows IDL programs to merge routines written in other languages with IDL at run-time. Each call to LINKIMAGE defines a new system procedure or function by specifying the routine's name, the name of the file containing the code, and the entry point name. The name of your routine is added to IDL's internal system routine table, making it available in the same manner as any other IDL built-in routine.

LINKIMAGE is the easiest way to add your system routines to IDL. It does not require linking a separate version of the IDL program with your code the way a direct call to **IDL_SysRtnAdd()** does, and it does not require writing the extra code required for a Dynamically Loadable Module (DLM). It is therefore commonly used for simple applications, and for testing during the development of a system routine.

If you are developing a larger application, or if you intend to redistribute your work, you should package your routines as Dynamically Loadable Modules, which are much easier for end-users to install and use than LINKIMAGE calls. You will find that the small additional programming effort is more than repaid from the time saved providing support for your code to your users.

If your IDL application relies on code written in languages other than IDL and linked into IDL using the LINKIMAGE procedure, you must make sure that the routines declared with LINKIMAGE are linked into IDL before any code that calls them is restored. In practice, the best way to do this is to make the calls to LINKIMAGE in your MAIN procedure, and include the code that uses the linked routines in a secondary `.SAV` file. In this case your MAIN procedure may look something like this:

```
PRO main

;Link the external code.
LINKIMAGE, 'link_function', 'new.dll'

;Restore code that uses linked code.
RESTORE, 'secondary.sav'

;Run your application.
myapp

END
```

In this scenario, the IDL code that calls the LINK_FUNCTION routine (the routine linked into IDL in the LINKIMAGE call) is contained in the secondary `.SAV` file `'secondary.sav'`.

Note

When creating your secondary `.SAV` file, you will need to issue the LINKIMAGE command before calling the SAVE procedure to link your routine into IDL after you have exited and restarted. The RESOLVE_ALL routine does not resolve routines linked to IDL with the LINKIMAGE procedure.

Dynamically Loadable Modules do not have this issue, and are the best way to avoid the problem.

Dynamically Loadable Modules

LINKIMAGE can be used to make IDL load your system routines in a simple and efficient manner. However, it quickly becomes inconvenient if you are adding more than a few routines. Furthermore, the limitation that the LINKIMAGE call must happen before any code that calls it is compiled makes it difficult to use and complicates the process of redistributing your routines to others. IDL offers an alternative method of packaging your system routines, called Dynamically Loadable Modules (DLMs), that address these and other problems.

The IDL_SYSFUN_DEF2 structure, which is described in "Registering Routines" on page 438, contains all the information required by IDL for it to be able to compile calls to a given system routine and call it:

- A routine signature (Name, minimum and maximum number of arguments, if the routine accepts keywords).
- A pointer to a compiled language function (usually C) that supplies the standard IDL system routine interface (argc, argv, argk) and which implements the desired operation.

IDL does not require the actual code that implements the function until the routine is called: It is able to compile other routines and statements that reference it based only on its signature.

DLMs exploit this fact to load system routines on an "as needed" basis. The routines in a DLM are not loaded by IDL unless the user calls one of them. A DLM consists of two files:

1. A module description file (human readable text) that IDL reads when it starts running. This file tells IDL the signature for all system routines contained in the loadable module.
2. A sharable library that implements the actual system routines.This library must be coded to present a specific IDL mandated interface (described below) that allows IDL to automatically load it when necessary without user intervention.

DLMs are a powerful way to extend IDL's built in system routines. This form of packaging offers many advantages:

- Unlike LINKIMAGE, IDL automatically discovers DLMs when it starts up without any user intervention. This makes them easy to install — you simply copy the two files into a directory on your system where IDL will look for them.

- DLM routines work exactly like standard built in routines, and are indistinguishable from them. There is no need for the user to load them (for example, using LINKIMAGE) before compiling code that references them.
- As the amount of code added to IDL grows, using sharable libraries in this way prevents name collisions in unrelated compiled code from fooling the linker into linking the wrong code together. DLMs thus act as a firewall between unrelated code. For example, there are instances where unrelated routines both use a common third party library, but they require different versions of this library. A specific example is that the HDF support in IDL requires its own version of the NetCDF library. The NetCDF support uses a different incompatible version of this library with the same names. Use of DLMs allows each module to link with its own private copy of such code.
- Since DLMs are separate from the IDL program, they can be built and distributed on their own schedule independent of IDL releases.
- System routines packaged as DLMs are effectively indistinguishable from routines built into IDL by RSI.

Use of sharable libraries in this manner has ample precedent in the computer industry. Most modern operating systems use loadable kernel modules to keep the kernel small while the functionality grows. The same technique is used in user programs in the form of sharable libraries, which allows unrelated programs to share code and memory space (e.g. a single copy of the C runtime library is used by all running programs on a given system).

How DLMs Work

IDL manages DLMs in the following manner:

1. When IDL starts, it looks in the current working directory for module definition (.dlm) files. It reads any file found and adds the routines and structure definitions thus defined to its internal routine and structure lookup tables as “stubs”. In the system routine dispatch table, stubs are entries that inform IDL of the routines existence, but which lack an actual compiled function to call. They contain sufficient information for IDL to properly compile calls to the routines, but not to actually call them. Similarly, stub entries in the structure definition table allow IDL to know that the DLM supplies the structure definition, but the actual definition is not present.

   After the current working directory, IDL searches !DLM_PATH for .dlm files and adds them to the table in the same manner. The default value of !DLM_PATH is the directory in the IDL distribution where the binary executables are kept. This default can be changed by defining the IDL_DLM_PATH environment variable (similarly to the way the IDL_PATH environment variable works with !PATH). This process happens once at startup, and never again. This means that IDL's knowledge of loadable modules is static and unchangeable once the session is underway. This is very different from the way !PATH works, and reflects the static nature of built in routines. The format of .dlm files is discussed in "The Module Description File" on page 452.

2. The IDL session then continues in the usual fashion until a call to a routine from a loadable module occurs. At that time, the IDL interpreter notices the fact that the routine is a stub, and loads the sharable library for the loadable module that supplies the routine. It then looks up and calls a function named **IDL_Load(),** which is required to exist, from the library. It's job is to replace the stubs from that module with real entries (by using **IDL_SysRtnAdd**()) and otherwise prepare the module for use.
3. Once the module is loaded, the interpreter looks up the routine that caused the load one more time. If it is still a stub then the module has failed to load properly and an error is issued. Normally, a full routine entry is found and the interpreter successfully calls the routine.
4. At this point the module is fully loaded, and cannot be distinguished from a compiled in part of IDL. A module is only loaded once, and additional calls to any routine, or access to any structure definition, from the module are made immediately and without requiring any additional loading.

The Module Description File

The module description file is a simple text file that is read by IDL when it starts. The information in this file tells IDL everything it needs to know about the routines supplied by a loadable module. With this information, IDL can compile calls to these routines and otherwise behave as if it contains the actual routine. The loadable module itself remains unloaded until a call to one of its routines is made, or until the user forces the module to load by calling the IDL DLM_LOAD procedure.

Empty lines are allowed in .dlm files. Comments are indicated using the # character. All text from a # to the end of the line is ignored by IDL and is for the user's benefit only.

All other lines start with a keyword indicating the type of information being conveyed, possibly followed by arguments. The syntax of each line depends on the keyword. Possible lines are:

MODULE Name

Gives the name of the DLM. This should always be the first non-comment line in a .dlm file.There can only be one MODULE line.

MODULE JPEG

DESCRIPTION DescriptiveText

Supplies a short one line description of the purpose of the module. This information is displayed by HELP,/DLM. This line is optional.

DESCRIPTION IDL JPEG support

VERSION VersionString

Supplies a version string that can be used by the IDL user to determine which version of the module will be used. IDL does not interpret this string, it only displays it as part of the **HELP,/DLM** output. This line is optional.

VERSION 6a

BUILD_DATE DateString

If present, IDL will display this information as part of the output from HELP,/DLM. IDL does not parse this string to determine the date, it is simply for the users benefit. This line is optional.

BUILD_DATE JAN 8 1998

SOURCE SourceString

A short one line description of the person or organization that is supplying the module. This line is optional.

SOURCE Research Systems, Inc.

CHECKSUM CheckSumValue

This directive is used by RSI to sign the authenticity of the DLMs supplied with IDL releases. It is not required for user-written DLMs.

STRUCTURE StructureName

There should be one STRUCTURE line in the DLM file for every named structure definition supplied by the loadable module. If you refer to such a structure before the

DLM is loaded, IDL uses this information to cause the DLM to load. The **IDL_Init()** function for the DLM will define the structure.

FUNCTION RtnName [MinArgs] [MaxArgs] [Options...]

PROCEDURE RtnName [MinArgs] [MaxArgs] [Options...]

There should be one FUNCTION or PROCEDURE line in the DLM file for every IDL routine supplied by the loadable module. These lines give IDL the information it needs to compile calls to these routines before the module is loaded.

RtnName

The IDL user level name for the routine.

MinArgs

The minimum number of arguments accepted by this routine. If not supplied, 0 is assumed.

MaxArgs

The maximum number of arguments accepted by this routine. If not supplied, 0 is assumed.

Options

Zero or more of the following:

OBSOLETE

IDL should issue a warning message if this routine is called and **!WARN.OBS_ROUTINE** is set.

KEYWORDS

This routine accepts keywords as well as plain arguments.

```
PROCEDURE    READ_JPEG   1   3    KEYWORDS
```

The IDL_Load() function

Every loadable module sharable library must export a single symbol called **IDL_Load().** This function is called when IDL loads the module, and is expected to do all the work required to load real definitions for the routines supplied by the function and prepare the module for use. This always requires at least one call to

IDL_SysRtnAdd(). It usually also requires a call to **IDL_MessageDefineBlock()** if the module defines any messages. Any other initialization needed would also go here:

```
int IDL_Load(void)
```

This function takes no arguments. It is expected to return *True* (non-zero) if it was successful, and *False* (0) if some initialization step failed.

DLM Example

This example creates a loadable module named **TESTMODULE**. **TESTMODULE** provides 2 routines:

TESTFUN

A function that issues a message indicating that it was called, and then returns the string “TESTFUN” This function accepts between 0 and **IDL_MAXPARAMS** arguments, but it does not use them for anything.

TESTPRO

A procedure that issues a message indicating that it was called. This procedure accepts between 0 and **IDL_MAX_ARRAY_DIM** arguments, but it does not use them for anything.

The intent of this example is to show the support code required to write a DLM for a completely trivial application. This framework can be easily adapted to real modules by replacing TESTFUN and TESTPRO with other routines.

The first step is to create the module definition file for TESTMODULE, named testmodule.dlm:

```
MODULE testmodule
DESCRIPTION Test code for loadable modules
VERSION 1.0
SOURCE Research Systems, Inc.
BUILD_DATE JAN  8 1998
FUNCTION TESTFUN 0 IDL_MAXPARAMS
PROCEDURE TESTPRO 0 IDL_MAX_ARRAY_DIM
```

The next step is to write the code for the sharable library. The contents of testmodule.c is shown in the following figure. Comments in the code explain what each step is doing.

C

```c
#include <stdio.h>
#include "idl_export.h"

/* Define message codes and their corresponding printf(3) format
 * strings. Note that message codes start at zero and each one is
 * one less that the previous one. Codes must be monotonic and
 * contiguous. */
static IDL_MSG_DEF msg_arr[] = {
#define M_TM_INPRO                       0
  {  "M_TM_INPRO",   "%NThis is from a loadable module procedure.” },
#define M_TM_INFUN                       -1
  {  "M_TM_INFUN”,   "%NThis is from a loadable module function.” },
};

/* The load function fills in this message block handle with the
 * opaque handle to the message block used for this module. The other
 * routines can then use it to throw errors from this block. */
static IDL_MSG_BLOCK msg_block;

/* Implementation of the TESTPRO IDL procedure */
static void testpro(int argc, IDL_VPTR *argv)
{ IDL_MessageFromBlock(msg_block, M_TM_INPRO, IDL_MSG_RET); }

/* Implementation of the TESTFUN IDL function */
static IDL_VPTR testfun(int argc, IDL_VPTR *argv)
{
  IDL_MessageFromBlock(msg_block, M_TM_INFUN, IDL_MSG_RET);
  return IDL_StrToSTRING("TESTFUN");
}

int IDL_Load(void)
{
  /* These tables contain information on the functions and procedures
   * that make up the TESTMODULE DLM. The information contained in these
   * tables must be identical to that contained in testmodule.dlm.
   */
  static IDL_SYSFUN_DEF2 function_addr[] = {
    { testfun, "TESTFUN”, 0, IDL_MAXPARAMS, 0, 0},
  };
  static IDL_SYSFUN_DEF2 procedure_addr[] = {
    { (IDL_SYSTRN_GENERIC) testpro, "TESTPRO”, 0, IDL_MAX_ARRAY_DIM, 0, 0},
  };

  /* Create a message block to hold our messages. Save its handle where
   * the other routines can access it. */
  if (!(msg_block = IDL_MessageDefineBlock("Testmodule”,
                                           IDL_CARRAY_ELTS(msg_arr),
                                           msg_arr))) return IDL_FALSE;

  /* Register our routine. The routines must be specified exactly the same
   * as in testmodule.dlm. */
  return IDL_SysRtnAdd(function_addr, TRUE,
                       IDL_CARRAY_ELTS(function_addr))
    && IDL_SysRtnAdd(procedure_addr, FALSE,
                     IDL_CARRAY_ELTS(procedure_addr));
}
```

Table 21-7: testmodule.c

If building a DLM for Microsoft Windows, a linker definition file (testmodule.def) is also needed. All of these files, along with the commands required to build the module can be found in the dlm subdirectory of the external directory of the IDL distribution.

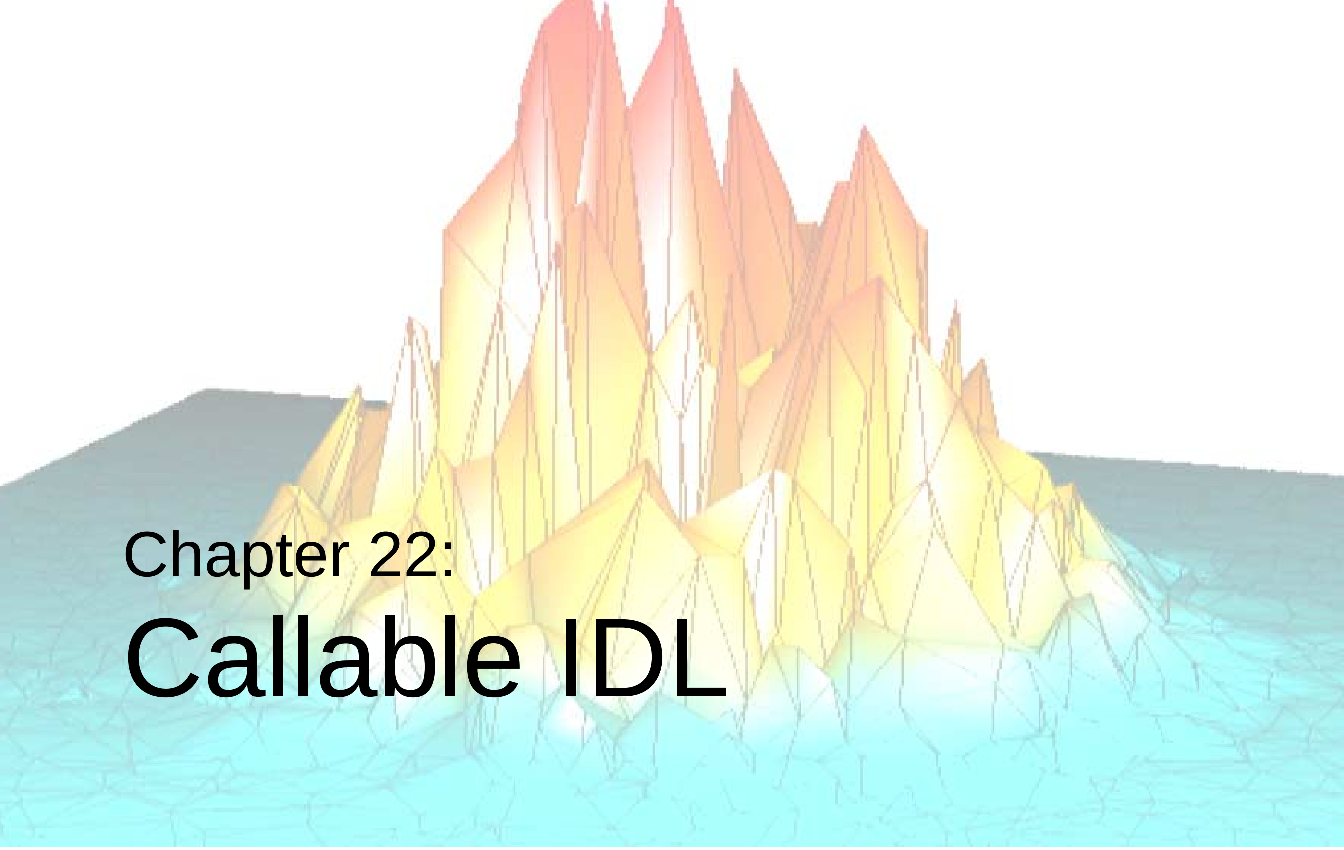
Once the loadable module is built, you can cause IDL to find it by doing one of the following:

- Move to the directory containing the .dlm and sharable library for the module.
- Define the IDL_DLM_PATH environment variable to include the directory.

Running IDL to demonstrate the resulting module:

```
IDL> HELP,/DLM,’testmodule’
** TESTMODULE - Test code for loadable modules (not loaded)
Version:1.0,Build Date:JAN 8 1998,Source:ResearchSystems, Inc.
Path: /home/user/testmodule/external/testmodule.so
IDL> testpro
% Loaded DLM: TESTMODULE.
% TESTPRO: This is from a loadable module procedure.
IDL> HELP,/DLM,’testmodule’
** TESTMODULE - Test code for loadable modules (loaded)
Version:1.0,Build Date:JAN 8 1998,Source:ResearchSystems, Inc.
Path: /home/user/testmodule/external/testmodule.so
IDL> print, testfun()
% TESTFUN: This is from a loadable module function.
TESTFUN
```

The initial HELP output shows that the module starts out unloaded. The call to TESTPRO causes the module to be loaded. As IDL loads the module, it prints an announcement of the fact (similar to the way it announces the .pro files it automatically compiles to satisfy calls to user routines). Once the module is loaded, subsequent calls to HELP show that it is present. Calls to routines from this module do not cause the module to be reloaded (as evidenced by the fact that calling TESTFUN did not cause an announcement message to be issued).



Chapter 22: Callable IDL

This chapter discusses the following topics:

Calling IDL as a Subroutine

IDL can be called as a subroutine from other programs. This capability is referred to as Callable IDL to distinguish it from the more common case of calling your code from IDL (as with CALL_EXTERNAL or as a system routine (LINKIMAGE, Dynamically Loadable Module)).

How Callable IDL is Implemented

IDL is built in a sharable form that allows other programs to call IDL as a subroutine. The specific details of how IDL is packaged depend on the platform:

- IDL for UNIX has a small driver program linked to a sharable object library that contains the actual IDL program.
- IDL for Windows consists of a driver program that implements the user interface (known as the IDE) linked to a dynamic-link library (DLL) that contains the actual IDL program.

In all cases, it is possible to link the sharable portion of IDL into your own programs. Note that Callable IDL is not a separate copy of IDL that implements a library version of IDL. It is in fact the same code, being used in a different context.

When is Callable IDL Appropriate?

Although Callable IDL is very powerful and convenient, it is not always the best method of communication between IDL and other programs. There are usually easier approaches that will solve a given problem. See "Supported Inter-Language Communication Techniques in IDL" on page 13 for alternatives.

IDL will not integrate with *all* programs. Understanding the issues described in this section will help you decide when Callable IDL is and is not appropriate.

Technical Issues Relating to Callable IDL

IDL makes computing easier by raising the level at which IDL users interface with the computer. It is natural to think that calling IDL from other programs will have the same effect, and under the correct circumstances this is true. However, using Callable IDL is not as easy as using IDL. Programmers who wish to use Callable IDL need to possess the skills described in "Skills Required to Combine External Code with IDL" on page 23.

Be aware that the same things that make IDL powerful at the user level can make it difficult to include in other programs. As an interactive, interpreted language, IDL is a decidedly non-trivial object to add to a process. Unlike a simple mathematical subroutine, IDL includes a compiler, a language interpreter, and related code that the caller must work around. As an interactive program, IDL must control the process to a high degree, which can conflict with the caller's wishes. The following (certainly incomplete) list summarizes some of the issues that must be dealt with.

UNIX IDL Signal API

IDL uses UNIX signals to manage many of its features, including exception handling, user interrupts, and child processes. The exact signals used and the manner in which they are used can change from IDL release to release as necessary. Although the IDL signal API (described in "IDL Internals: UNIX Signals" on page 353) allows you to use signals in an IDL-compatible way, the resulting constraints may require changes to your code.

IDL Timer API

IDL's use of the process timer requires you to use the IDL timer API instead of the standard system routines. This restriction may require changes to some programs. Under UNIX, the timer module can interrupt system calls. Timers are discussed in "IDL Internals: Timers" on page 387.

GUI Considerations

Most applications will call IDL and display IDL graphics in an IDL window. However, programmers may want to write applications in which they create the graphical user interface (GUI) and then have IDL draw graphics into windows that IDL did not create. It is not always possible for IDL to draw into windows that it did not create for the reasons described below:

X Windows

The IDL X Windows graphics driver can draw in windows it did not create as long as the window is compatible with the IDL display connection (see Appendix A, “IDL Graphics Devices” in the *IDL Reference Guide* manual for details). However, the design of IDL’s X Windows driver requires that it open its own display connection and run its own event loop. If your program cannot support a separate display connection, or if dividing time between two event loops is not acceptable, consider the following options:

- Run IDL in a separate process and use interprocess communication (possibly Remote Procedure Calls, to control it.
- If you choose to use Callable IDL, use the IDL Widget stub interface, described in “Adding External Widgets to IDL” on page 501, to obtain the IDL display connection, and create your GUI using that connection rather than creating your own. The IDL event loop will dispatch your events along with IDL’s, creating a well-integrated system.

Microsoft Windows

At this time, the IDL for Windows graphics driver does not have the ability to draw into windows that were not created by IDL. However, the ActiveX control described in Chapter 6, “The IDLDrawWidget ActiveX Control”, can do this.

Program Size Considerations

On systems that support preemptive multitasking, a single huge program is a poor use of system capabilities. Such programs inevitably end up implementing primitive task-scheduling mechanisms better left to the operating system.

Troubleshooting

Troubleshooting and debugging applications that call IDL can be very difficult. With standard IDL, malfunctions in the program are clearly the fault of RSI, and given a reproducible bug report, we attempt to fix them promptly. A program that combines IDL with other code makes it difficult to unambiguously determine where the problem lies. The level of support RSI can provide in such troubleshooting is minimal. The programmer is responsible for locating the source of the difficulty. If the problem is in IDL, a simple program demonstrating the problem must be provided before we can address the issue.

Threading

IDL uses threads to implement its thread pool functionality, which is used to speed numerical computation on multi-CPU hardware. Despite this, it is essentially a single threaded program, and is not designed to be called from different threads of a threaded application. Attempting to use IDL from any thread other than the main thread is unsupported, and may cause unpredictable results.

Inter-language Calling Conventions

IDL is written in standard ANSI C. Calling it from other languages is possible, but it is the programmer's responsibility to understand the inter-language calling conventions of the target machine and compiler.

Appropriate Applications of Callable IDL

Callable IDL is most appropriate in the following situations:

- Callable IDL is clearly the correct choice when the resulting program is to be a front-end that creates a different interface for IDL. For example, you might wish to turn IDL into an RPC server that uses an RPC protocol not directly supported by IDL, or use IDL as a module in a distributed system.
- Callable IDL is appropriate if either the calling program or IDL handles *all* graphics, including the Graphical User Interface, *without the involvement of the other*. Intermediate situations are possible, but more difficult. In particular, beware of attempts to have two event/message loops.
- Callable IDL is appropriate when the calling program makes little or no use of signals, timers, or exception handling, or is able to operate within the constraints imposed by IDL.

Licensing Issues and Callable IDL

If you intend to distribute an application that calls IDL, note that each copy of your application must have access to a properly licensed copy of the IDL library. For availability of a runtime version of IDL, contact RSI or your IDL distributor.

Using Callable IDL

The process of using Callable IDL has three stages: initialization, IDL use, and cleanup. Between the initialization and the cleanup, your program contains a complete active IDL session, just as if a user were typing commands at an `IDL>` prompt. In addition to the usual IDL abilities, you can import data from your program and cause IDL to see it as an IDL variable. IDL can use such data in computations as if it had created the variable itself. In addition, you can obtain pointers to data currently held by IDL variables and access the results of IDL computations from your program.

Note

The functions documented in this chapter should only be used when calling IDL from other programs—their use in code called by IDL via CALL_EXTERNAL or a system routine (LINKIMAGE, Dynamically Loadable Module) is not supported and is certain to corrupt and/or crash the IDL process.

Before calling IDL to execute instructions, you must initialize it. Under UNIX, you do this by calling **IDL_Init().** Under Microsoft Windows, you call **IDL_Win32Init()** instead. This is a one-time operation, and must occur before calling any other IDL function. see "Initialization" on page 467 for complete information on this topic. Once IDL is initialized, you can:

1. Send IDL commands to IDL for execution. Commands are sent as strings, using the same syntax as interactive IDL. Note that there is not a separate C language function for every IDL command—any valid IDL command can be executed as IDL statements. This approach allows us to keep the callable IDL API small and simple while allowing full access to IDL's abilities. This is explained in "Executing IDL Statements" on page 473.
2. Call any of the several routines that interact with IDL through other means to perform operations such as:
   - Importing data into IDL. (See "Creating an Array from Existing Data" on page 286.)
   - Accessing data within IDL. (See "Looking Up Variables in Current Scope" on page 296.)
   - Changing items in the process, such as signal handling or timers. (See "IDL Internals: UNIX Signals" on page 353, or "IDL Internals: Timers" on page 387.)

- Redirecting IDL output to your own function for processing. See “Diverting IDL Output” on page 471.

The above list is not complete, but is representative of the possibilities afforded by Callable IDL.

Cleanup

After all IDL use is complete, but before the program exits, you must call **IDL_Cleanup()** to allow IDL to shutdown gracefully and clean up after itself. Once this has been done, you are not allowed to call IDL again from this process. See “Cleanup” on page 475.

Initialization

IDL for UNIX uses the **IDL_Init()** function (described below) to prepare Callable IDL for use. IDL for Microsoft Windows uses **IDL_Win32Init()**, described in "Initialization: Microsoft Windows" on page 469.

Initialization: UNIX

IDL for UNIX uses the **IDL_Init()** function prepares Callable IDL for use. This must be the first IDL routine called.

Note

Microsoft Windows applications should not call **IDL_Init()**. Instead, use **IDL_Win32Init(), d**escribed in "Initialization: Microsoft Windows" on page 469.

```
int IDL_Init(int options, int *argc, char *argv[]);
```

where:

options

A bitmask used to specify initialization options. The allowed bit values are:

IDL_INIT_EMBEDDED

Setting this bit causes IDL to initialize to run applications from a Save/Restore file that contains an embedded license. **IDL_RuntimeExec()** is then used to run the application(s). Note that **IDL_Execute()** and **IDL_ExecuteStr()** are disabled when IDL is initialized with this option.

IDL_INIT_GUI

Setting this bit causes IDL to use the IDL Development Environment (IDLDE) GUI rather than using the standard tty based interface. This option is ignored under Microsoft Windows.

IDL_INIT_GUI_AUTO

Setting this bit causes IDL to try to use the IDL Development Environment (IDLDE) GUI. If that fails, IDL uses the standard tty interface. This option is ignored under Microsoft Windows.

IDL_INIT_LMQUEUE

Setting this bit causes IDL to wait for an available license before beginning an IDL task such as batch processing.

IDL_INIT_NOLICALIAS

Our FLEXlm floating licence policy is to alias all IDL sessions that share the same user/system/display to the same license. If IDL_INIT_NOLICALIAS is set, this IDL session will force a unique license to be checked out. In this case, we allow the user to change the DISPLAY environment variable. This is useful for RPC servers that don't know where their output will need to go before invocation.

IDL_INIT_BACKGROUND (IDL_INIT_NOTTYEDIT)

Indicates to IDL that it is going to be used in a background mode by some other program, and that IDL will not be in control of the user's input command processing.

One effect of this is that XMANAGER will realize that the active command line functionality for processing widget events is not available, and XMANAGER will block to manage events when it is called rather than return immediately.

Normally under UNIX, if IDL sees that **stdin** and **stdout** are ttys, it puts the tty into raw mode and uses termcap/terminfo to handle command line editing. When using callable IDL in a background process that isn't doing input/output to the tty, the termcap initialization can cause the process to block (because of job control from the shell) with a message like "Stopped (tty output) idl". Setting this option prevents all tty edit functions and disables the calls to termcap. I/O to the tty is then done with a simple **fgets()/printf()**. If the **IDL_INIT_GUI** bit is set, this option is ignored.

For historical reasons, this option used to be called **IDL_INIT_NOTTYEDIT**. Use of that name is still supported.

IDL_INIT_QUIET

Setting this bit suppresses the display of the startup announcement and message of the day.

IDL_INIT_RUNTIME

Setting this bit causes IDL to check out a runtime license instead of the normal license. **IDL_RuntimeExec()** is then used to run an IDL application restored from a Save/Restore file.

argc

As passed by the operating system to **main()**.

argv

As passed by the operating system to **main()**.

IDL_Init() returns TRUE if the initialization is successful, and FALSE for failure. Arguments not directly intended for IDL are removed from **argv** and **argc** is decremented to match.

Initialization: Microsoft Windows

Under Microsoft Windows, the **IDL_Win32Init()** function prepares the IDL DLL for use. **IDL_Win32Init()** must be called before any other function except **IDL_ToutPush()**.

Note

Windows applications should not call **IDL_Init()**, described in the previous section. **IDL_Win32Init()** calls **IDL_Init()** on your behalf at the appropriate time.

```
int IDL_Win32Init(int iOpts, void *hinstExe, void *hwndExe,
                  void *hAccel);
```

where:

iOpts

A bitmask used to specify initialization options. The allowed bit values are:

IDL_INIT_RUNTIME

Setting this bit causes IDL to check out a runtime license instead of the normal license. **IDL_RuntimeExec()** is then used to run an IDL application restored from a Save/Restore file.

IDL_INIT_LMQUEUE

Setting this bit causes IDL to wait for an available license before beginning an IDL task such as batch processing.

hinstExe

HINSTANCE from the application that will be calling IDL.

hwndExe

HWND for the application's main window.

hAccel

Reserved. This argument should always be NULL.

IDL_Win32Init() returns TRUE if the initialization is successful, and FALSE for failure.

Diverting IDL Output

When using a tty-based interface (available only on UNIX platforms), IDL sends its output to the screen for the user to see. When using a GUI-based interface (any platform), the output goes to the IDL log window. The default output function is automatically installed by IDL at startup. To divert IDL output to a function of your own design, use **IDL_ToutPush()** and **IDL_ToutPop()** to change the output function called by IDL.

Internally, IDL maintains a stack of output functions, and provides two functions (**IDL_ToutPush()** and **IDL_ToutPop()**) to manage them. The most recently pushed output function is called to output each line of text. Output functions of your own design should have the following type definition:

```
typedef void (* IDL_TOUT_OUTF)(int flags, char *buf, int n);
```

The arguments to an output function are:

flags

A bitmask of flag values that specify how the text should be output. The allowed bit values are:

IDL_TOUT_F_STDERR

Send the text to **stderr** rather than **stdout**, if that distinction means anything to your output device.

IDL_TOUT_F_NLPOST

After outputting the text, start a new output line. On a tty, this is equivalent to sending a newline (`'\n'`) character.

buf

The text to be output. There may or may not be a NULL termination, so the character count provided by **n** must be used to move only the specified number of characters.

n

The number of characters in **buf** to be output.

IDL_ToutPush()

Use **IDL_ToutPush()** to push a new output function onto the stack. The most recently pushed function is the one used by IDL for output.

```
void IDL_ToutPush(IDL_TOUT_OUTF outf);
```

IDL_ToutPop()

IDL_ToutPop() removes the most recently pushed output function. The removed function pointer is returned.

```
IDL_TOUT_OUTF IDL_ToutPop(void);
```

Warning

Do not pop an output function you did not push. It is an error to attempt to remove the last remaining function.

Executing IDL Statements

There are two functions that allow you to execute IDL statements. **IDL_ExecuteStr()** executes a single command, while **IDL_Execute()** takes an array of commands and executes them in order. In both cases, the commands are null terminated strings—just as they would be typed by an IDL user at the `IDL>` prompt. It is important to realize that the full abilities of IDL are available at this point. Typically, the commands you issue will run IDL programs of varying complexity, including support routines written in IDL from the IDL Library (found via the IDL !PATH system variable). This ability to “download” complicated programs into IDL and then run them via a simple command can be very powerful.

IDL_Execute()

IDL_Execute() executes the command strings in the order given. It returns the value of !ERROR_STATE.CODE after the final command has executed. If the value of !ERROR_STATE.CODE is needed for an intermediate command, you should use **IDL_ExecuteStr()** instead of **IDL_Execute()**.

```
int IDL_Execute(int argc, char *argv[]);
```

argc

The number of commands contained in **argv**.

argv

An array of pointers to NULL-terminated strings containing IDL statements to execute.

IDL_ExecuteStr()

IDL_ExecuteStr() returns the value of the !ERROR_STATE.CODE system variable after the command has executed.

```
int IDL_ExecuteStr(char *cmd);
```

cmd

A NULL-terminated string containing an IDL statement to execute.

Runtime IDL and Embedded IDL

If you distribute programs that call IDL with a runtime license or an embedded license, use **IDL_RuntimeExec()**. After initialization **IDL_RuntimeExec()** can be used to run self-contained IDL applications from a Save/Restore file. **IDL_RuntimeExec()** restores the file, then attempts to call an IDL procedure named MAIN. If no MAIN procedure is found, the function attempts to call a procedure with the same name as the restored Save file. (That is, if the Save file is named `myprog.sav`, **IDL_RuntimeExec()** looks for a procedure named `myprog`.)

IDL_RuntimeExec() returns TRUE if the operation succeeded and the MAIN procedure or the named procedure were called. Note that the returned status *does not* indicate whether the actual IDL code ran successfully.

```
int IDL_RuntimeExec(char *file);
```

where:

file

The complete path specification to the Save file to be restored, in the native syntax of the platform in use.

Cleanup

After your program is finished using IDL (typically just before it exits) it should call **IDL_Cleanup()** to allow IDL to shut down gracefully. **IDL_Cleanup()** returns a status value that can be passed to **Exit()**.

```
int IDL_Cleanup(int just_cleanup);
```

where:

just_cleanup

If TRUE, **IDL_Cleanup()** does all the process shutdown tasks, but doesn't actually exit the process. If FALSE (the usual), the process exits.

Microsoft Windows applications should place this call in their Main **WndProc** to be called as a result of the **WM_CLOSE** message.

```
switch(msg){
           ...
      case WM_CLOSE:
           IDL_Cleanup(TRUE);
           any additional processing
           ...
```

Issues and Examples: UNIX

Interactive IDL

Under UNIX, **IDL_Main()** implements IDL as seen by the interactive user. In the interactive version of IDL as shipped by RSI, the actual **main()** function simply decodes its arguments to determine which options to specify and then calls **IDL_Main()** to do the rest. **IDL_Main()** calls **exit()** and does not return to its caller.

```
int IDL_Main(int init_options, int argc, char *argv[]);
```

where:

init_options

The options argument to be passed to **IDL_Init()**.

argc, argv

From **main()**. Arguments that correspond to options specified via the **init_options** argument should be removed and converted to **init_options** flags prior to calling this routine.

Compiling Programs That Call IDL

A complete discussion of the issues that arise when compiling and linking C programs is beyond the scope of this manual. The following is a brief list of basic concepts to consider when building programs that call IDL.

- Compilers for some languages add underscores at the beginning or end of user defined names. To check the naming convention employed by your compiler, use the UNIX `nm(1)` command to list the symbols exported from an object file.

  If you use only one language, naming details are handled transparently by the compiler, linker, and debugger. If you use more than one language, problems can arise if the different compilers use different naming conventions. For example, the Fortran compiler might add an underscore to the end of each name, while the C compiler does not. To call a Fortran routine from C, you must then include this underscore in your code (to call the function `my_code`, you would refer to it as `my_code_`). Note that you may also need to set a compiler flag to make case significant.

To determine whether your compilers use compatible naming conventions, consult your compiler documentation or experiment with small test programs using the compilers and the `nm` command.

- Every program starts execution at a known routine. In the C language, this routine is explicitly named **main()**. In Fortran, execution begins with the implicit main program. If you are using Callable IDL, you must provide a **main()** function for your program.
- When linking a C program, use the `cc` command instead of the `ld` command. `cc` calls `ld` to perform the link operation, and when necessary adds a directive to `ld` that causes the C runtime library to be used.

  If you don't use `cc` to link your program (if you are using `ld` directly or are using a Fortran compiler, for example) and you get "unsatisfied symbol" errors for symbols that are in the standard C library, try including the runtime library explicitly in your link command. Usually, adding the string `-lc` to the end of the command is all that is necessary.

  Under Hewlett-Packard's HP-UX operating system, if you use `ld` directly you may also need to include the `PA1.1` math library in order to locate mathematics routines at runtime. Add the flag `-L/lib/pa1.1` prior to `-lm` on the link line to link with the `PA1.1` math libraries.

  See "Compilation And Linking Details" on page 31 for advice on how to compile and link programs with the IDL libraries under various operating systems.

Example: Calling IDL From C

The program in the following figure(`calltest.c`, found in the `callable` subdirectory of the `external` subdirectory of the IDL distribution) demonstrates how to import data from a C program into IDL, execute IDL statements, and obtain data from IDL variables. It performs the following actions:

1. Create an array of 10 floating point values with each element set to the value of its index. This is equivalent to the IDL command FINDGEN(10).
2. Initialize Callable IDL.
3. Import the floating point array into IDL as a variable named TMP.
4. Have IDL print the value of TMP.

5. Execute a short sequence of IDL statements from a string array:

```
tmp2 = total(tmp)
print,'IDL total is ',tmp2
plot, tmp
```

6. Set TMP to zero, causing IDL to release the pointer to the floating point array.
7. Obtain a pointer to the data contained in TMP2. From examining the IDL statements executed to this point, we know that TMP2 is a scalar floating point value.
8. From our C program, print the value of the IDL TMP2 variable.
9. Execute a small widget program. Pressing the button allows the program to end:

```
a = widget_base()
b = widget_button(a, value='Press When Done',xsize=300,
                  ysize=200)
widget_control, /realize, a
dummy = widget_event(a)
widget_control, /destroy, a
```

See “Compilation and Linking Statements” on page 490 for details on compiling and linking this program.

Each line is numbered to make discussion easier. The line numbers are not part of the actual program.

C

```
 1 #include <stdio.h>
 2 #include "idl_export.h"
 3
 4 static void free_callback(UCHAR *addr)
 5 {
 6    printf("IDL released(%u)\n", addr);
 7 }
 8
 9 int main(int argc, char **argv)
10 {
11   float f[10];
12   int i;
13   IDL_VPTR v;
14   IDL_MEMINT dim[IDL_MAX_ARRAY_DIM];
15   static char *cmds[] = { "tmp2 = total(tmp)",
16     "print,'IDL total is ',tmp2", "plot,tmp" };
17   static char *cmds2[] = { "a = widget_base()",
18     "b = widget_button(a, value='Press When Done', xsize=300, ysize=200)",
19     "widget_control,/realize, a",
20     "dummy = widget_event(a)",
21     "widget_control,/destroy, a" };
22
23
24   for (i=0; i < 10; i++) f[i] = (float) i;
25   if (IDL_Init(0, &argc, argv)) {
26     dim[0] = 10;
27     printf("ARRAY ADDRESS(%u)\n", f);
28     if (v=IDL_ImportNamedArray("TMP", 1, dim, IDL_TYP_FLOAT,
29                  (UCHAR *) f, free_callback, (void *) 0)) {
30     (void) IDL_ExecuteStr("print, tmp");
31     (void) IDL_Execute(sizeof(cmds)/sizeof(char *), cmds);
32     (void) IDL_ExecuteStr("print, 'Free the user memory'");
33     (void) IDL_ExecuteStr("tmp = 0");
34     if (v = IDL_FindNamedVariable("tmp2", IDL_FALSE))
35       printf("Program total is %f\n", v->value.f);
36       (void) IDL_Execute(sizeof(cmds2)/sizeof(char *), cmds2);
37       IDL_Cleanup(IDL_FALSE);   /* Don't return */
38     }
39   }
40
41  return 1;
42 }
```

Table 22-1: Calling IDL from C on UNIX

Following is commentary on this program, by line number:

24

C equivalent to IDL command “F = FINDGEN(10)”

25

Initialize IDL

26–29

Import C array **F** into IDL as a FLTARR vector named **TMP** with 10 elements. Note use of the callback argument **free_callback**. This function will be called when IDL is finished with the array **F**, giving us a chance to properly clean up at that time.

30

Have IDL print the value of **TMP**.

31

Execute the commands contained in the C string array **cmds** defined on lines 15–16. These commands create a new IDL variable named **TMP2** containing the sum of the elements of **TMP**, print its value, and plot the vector.

32–33

Set **TMP** to a new value. This will cause IDL to release the user supplied memory from lines 26–29 and call **free_callback**.

34–35

From C, get a reference to the IDL variable **TMP2** and print its value. This should agree with the value printed by IDL on line 31. It is important to realize that the pointer to the variable or anything it points at can only be used until the next call to execute an IDL statement. After that, the pointer and the contents of the referenced **IDL_VARIABLE** may become invalid as a result of IDL's execution.

36

Run the simple IDL widget program contained in the array C string array **cmds2** defined on lines 17–21.

37

Shut down IDL. The IDL_FALSE argument instructs **IDL_Cleanup()** to exit the process, so this call should not return.

41

This line should never be reached. If it is, return the UNIX failing status.

Example: Calling an IDL Math Function

This example demonstrates how to write a simple C wrapper function that allows calling IDL commands simply from another language. We implement a function named **call_idl_fft()** that calls the IDL FFT function operating on data imported from our C program. It returns TRUE on success, FALSE for failure:

```
int call_idl_fft(IDL_COMPLEX *data, int n, int direction);
```

data

A pointer to a linear array of complex data to be processed.

n

The number of data points contained in the array data.

dir

The direction of the FFT transform to take. Specify **-1** for a forward transform, **1** for the reverse

The program is shown in the following figure. Each line is numbered to make discussion easier. These numbers are not part of the actual program. Following is commentary on the above program, by line number:

C

```
 1 #include <stdio.h>
 2 #include "idl_export.h"
 3
 4
 5 int call_idl_fft(IDL_COMPLEX *data, IDL_MEMINT n, int dir)
 6 {
 7   int r;
 8   IDL_MEMINT dim[IDL_MAX_ARRAY_DIM];
 9   char buf[64];
10
11   dim[0] = n;
12   if (IDL_ImportNamedArray("TMP_FFT_DATA", 1, dim,
13       IDL_TYP_COMPLEX, (UCHAR *) data, 0, 0)) {
14     (void) IDL_ExecuteStr("MESSAGE, /RESET");
15     sprintf(buf,"TMP_FFT_DATA=FFT(TMP_FFT_DATA,/OVERWRITE)"
16            ,dir);
17     r = !IDL_ExecuteStr(buf);
18     (void) IDL_ExecuteStr("TMP_FFT_DATA=0");
19   } else {
20     r = FALSE;
21   }
22
23   return r;
24 }
25
26 main(int argc, char **argv)
27 {
28 #define NUM_PNTS 10
29   IDL_COMPLEX data[NUM_PNTS];
30   int i;
31
32   for (i = 0; i < NUM_PNTS; i++) data[i].r = data[i].i = i;
33   if (IDL_Init(0, &argc, argv)) {
34     call_idl_fft(data, NUM_PNTS, -1);
35     call_idl_fft(data, NUM_PNTS, 1);
36     for (i = 0; i < NUM_PNTS; i++)
37       printf("(%f, %f)\n", data[i].r, data[i].i);
38     IDL_Cleanup(IDL_FALSE);
39   }
40
41   return 1;
42 }
```

Table 22-2: call_idl_fft()

7

The variable **r** holds the result from the function.

8

dim is used to import the data into IDL as an array.

9

A temporary buffer to format the IDL FFT command.

11–13

Import data into IDL as the variable **TMP_FFT_DATA**. We don’t set up a **free_callback** because we will explicitly force IDL to release the pointer after the call to FFT.

14

Set the !ERROR_STATE system variable back to the “success” state so previous errors don’t confuse our results.

15–16

Format an FFT command to IDL into **buf**. Note the use of the OVERWRITE keyword. This tells the IDL FFT function to place the results into the input variable rather than creating a separate output variable. Hence, the results end up in our data array without the need to obtain a pointer to the results and copy them out.

17

Have IDL execute the FFT statement. **IDL_ExecuteStr()** returns the value of !ERROR_STATE.CODE, which should be zero for success and non-zero in case of error. Hence, negating the result of **IDL_ExecuteStr()** yields the status value we require for the result of this function.

18

Set **TMP_FFT_DATA** to 0 within IDL. This causes IDL to release the data pointer imported previously.

20

If the call to **IDL_ImportNamedArray()** fails, we must report failure.

26

In order to test the **call_idl_fft()** function, this main program calls it twice. Taking numerical error into account the end result should be equal to the original data.

32

Set the real and imaginary part of each element to the index value.

33

Initialize Callable IDL.

34

Call **call_idl_fft()** to perform a forward transform.

35

Call **call_idl_fft()** to perform a reverse transform.

36–37

Print the results.

38

Shut down IDL and exit the process.

41

This line should never be reached. If it is, return the UNIX failing status.

Example: Calling IDL from Fortran

The program shown in the following figure (`CALLTEST`, found in the `callable` subdirectory of the `external` subdirectory of the IDL distribution) demonstrates how to import data from a Fortran program into IDL, execute IDL statements, and obtain data from IDL variables. See “Compilation and Linking Statements” on page 490 for details on compiling and linking this program. The source code for this file can be found in the file `calltest.f`, located in the `callable` subdirectory of the `external` subdirectory of the IDL distribution.

Each line is numbered to make discussion easier. The line numbers are not part of the actual program:

f77

```
C-------------------------------------------------------------------
C    Routine to print a floating point value from an IDL variable.

 SUBROUTINE PRINT_FLOAT(VPTR)

C    Declare a Fortran Record type that has a compatible form with
C    the IDL C struct IDL_VARIABLE for a floating point value.
C    Note this structure contains a union which is the size of
C    the largest data type. This structure has been padded to
C    support the union. Fortran records are not part of
C    F77, but most compilers have this option.

 STRUCTURE /IDL_VARIABLE/
           CHARACTER*1 TYPE
           CHARACTER*1 FLAGS
           INTEGER*4 PAD       !Pad for largest data type
           REAL*4 VALUE_F
 END STRUCTURE

 RECORD /IDL_VARIABLE/ VPTR

 WRITE(*, 10) VPTR.VALUE_F
  10 FORMAT('Program total is: ', F6.2)

 RETURN

 END

C-------------------------------------------------------------------
C   This function will be called when IDL is finished with the
C   array F.

        SUBROUTINE FREE_CALLBACK(ADDR)

           INTEGER*4 ADDR

           WRITE(*,20) LOC(ADDR)
  20    FORMAT ('IDL Released:', I12)

           RETURN

        END

C-------------------------------------------------------------------
C  This program demonstrates how to import data from a Fortran
C  program into IDL, execute IDL statements and obtain data
C  from IDL variables.

```

Table 22-3: Calling IDL from Fortran On UNIX

f77

```
 PROGRAM CALLTEST

C  Some Fortran compilers require external defs. for IDL routines:
        EXTERNAL IDL_Init !$pragma C(IDL_Init)
        EXTERNAL IDL_Cleanup !$pragma C(IDL_Cleanup)
        EXTERNAL IDL_Execute !$pragma C(IDL_Execute)
        EXTERNAL IDL_ExecuteStr !$pragma C(IDL_ExecuteStr)
        EXTERNAL IDL_ImportNamedArray !$pragma C(IDL_ImportNamedArray)
        EXTERNAL IDL_FindNamedVariable !$pragma C(IDL_FindNamedVariable)

C  Define arguments for IDL_Init routine
        INTEGER*4 ARGC
        INTEGER*4 ARGV(1)
        DATA ARGC, ARGV(1) /2 * 0/

C  Define IDL Definitions for IDL_ImportNamedArray

        PARAMETER (IDL_MAX_ARRAY_DIM = 8)
        PARAMETER (IDL_TYP_FLOAT = 4)

        REAL*4 F(10)
        INTEGER*4 DIM(IDL_MAX_ARRAY_DIM)
        DATA DIM /10, 7*0/
        INTEGER*4 FUNC_PTR     !Address of function
        INTEGER*4 VAR_PTR      !Address of IDL variable
        EXTERNAL FREE_CALLBACK !Declare ext routine for use as arg

        PARAMETER (MAXLEN=80)
        PARAMETER (N=10)

C  Define commands to be executed by IDL

        CHARACTER*(MAXLEN) CMDS(3)
        DATA CMDS /"tmp2 = total(tmp)",
     &            "print, 'IDL total is ', tmp2",
     &            "plot, tmp"/
        INTEGER*4 CMD_ARGV(10)

C  Define widget commands to be executed by IDL

        CHARACTER*(MAXLEN) WIDGET_CMDS(5)
        DATA  WIDGET_CMDS /"a = widget_base()",
   &    "b = widget_button(a,val='Press When Done',xs=300,ys=200)",
   &    "widget_control, /realize, a",
   &    "dummy = widget_event(a)",
   &    "widget_control, /destroy, a"/

        INTEGER*4 ISTAT

```

Table 22-3: (Continued) Calling IDL from Fortran On UNIX

f77

```
C  Null Terminate command strings and store the address
C  for each command string in CMD_ARGV

        DO I = 1, 3
           CMDS(I)(MAXLEN:MAXLEN) = CHAR(0)
           CMD_ARGV(I) = LOC(CMDS(I))
        ENDDO

C  Initialize floating point array, equivalent to IDL FINDGEN(10)

        DO I = 1, N
           F(I) = FLOAT(I-1)
        ENDDO

C  Print address of F

 WRITE(*,30) LOC(F)
   30 FORMAT('ARRAY ADDRESS:', I12)

C  Initialize Callable IDL

        ISTAT = IDL_Init(%VAL(0), ARGC, ARGV(1))

        IF (ISTAT .EQ. 1) THEN

C  Import the floating point array into IDL as a variable named TMP

        CALL IDL_ImportNamedArray('TMP'//CHAR(0), %VAL(1), DIM,
    &       %VAL(IDL_TYP_FLOAT), F, FREE_CALLBACK, %VAL(0))

C  Have IDL print the value of tmp

        CALL IDL_ExecuteStr('print, tmp'//CHAR(0))

C  Execute a short sequence of IDL statements from a string array

        CALL IDL_Execute(%VAL(3), CMD_ARGV)

C  Set tmp to zero, causing IDL to release the pointer to the
C  floating point array.

        CALL IDL_ExecuteStr('tmp = 0'//CHAR(0))

C  Obtain the address of the IDL variable containing the
C  the floating point data

        VAR_PTR = IDL_FindNamedVariable('tmp2'//CHAR(0), %VAL(0))

C  Call a Fortran routine to print the value of the IDL tmp2 variable
        CALL PRINT_FLOAT(%VAL(VAR_PTR))


```

Table 22-3: (Continued) Calling IDL from Fortran On UNIX

```
      150 C  Null Terminate command strings and store the address
      151 C  for each command string in CMD_ARGV
      152
      153          DO I = 1, 5
      154             WIDGET_CMDS(I)(MAXLEN:MAXLEN) = CHAR(0)
      155             CMD_ARGV(I) = LOC(WIDGET_CMDS(I))
      156          ENDDO
      157
      158 C  Execute a small widget program. Pressing the button allows
f77   159 C  the program to end
      160
      161          CALL IDL_Execute(%VAL(5), CMD_ARGV)
      162
      163 C  Shut down IDL
      164          CALL IDL_Cleanup(%VAL(0))
      165
      166        ENDIF
      167
      168    END
```

Table 22-3: (Continued) Calling IDL from Fortran On UNIX

1-27

In order to print variables returned from IDL, we must define a Fortran structure type for **IDL_VARIABLE**. This subroutine creates the **IDL_VARIABLE** structure and defines a way to print the floating-point value returned in the an IDL variable.

14-17

Define a Fortran structure equivalent to the floating-point portion of the C **IDL_VARIABLE** structure. Since we know our value is a floating-point number, only the floating-point portion of the structure is implemented. The structure is padded for the largest data type contained in the union. With some Fortran compilers, the combination of **UNION** and **MAP** can be used to implement the **ALLTYPES** union portion of the **IDL_VARIABLE** structure.

29-42

This subroutine is called when IDL releases the user-supplied memory.

44-164

This is the main Fortran program.

51-57

External definitions for IDL internal routines. These definitions may not be necessary with some Fortran compilers.

59-62

Define the **argc** and **argv** arguments required by **IDL_Init()**.

66-67

Define constants equivalent to C IDL constants for the maximum array dimensions and type **float**.

69-77

Define parameters necessary for **IDL_ImportNamedArray()**.

79-85

Define an array of IDL commands to be executed.

87-96

Define an array of IDL widget commands to be executed.

98-104

Null-terminate each of the command strings and store the address of each command to pass to IDL.

106-110

Initialize the floating-point array. This is the Fortran equivalent to the IDL command `F=FINDGEN(10)`.

117-121

Initialize IDL.

125-126

Import the Fortran array **F** in the IDL as a 10-element FLTARR vector named **TMP**. Note the use of the callback argument **FREE_CALLBACK()**, which will be called when IDL is finished with the array **F**, giving us a chance to clean up at that time.

134

Execute the commands contained in the character array **CMDS** defined on lines 71-77. The address for each command is stored in the corresponding array element of **CMD_ARGV**.

139

Set the **TMP** variable to a new value. This causes IDL to release the user-supplied memory and call **FREE_CALLBACK()**.

144

Get a reference to the IDL variable **TMP2**.

147

Call the routine **PRINT_FLOAT** to print the value of **TMP2**. This should agree with the value printed by line 130. Note that the address of the IDL variable **TMP2**, and its contents, can only be used until the next call to execute an IDL statement, since IDL may change the value of the referenced **IDL_VARIABLE**.

150-161

Execute the commands contained in the character array **WIDGET_CMDS** defined on lines 79-88.

163-168

Shut down IDL. The 0 argument instructs **IDL_CLEANUP()** to exit the process, so this call should not return.

Compilation and Linking Statements

Compilation and linking procedures used when calling IDL on a UNIX system are described in the file `calltest_unix.txt` in the `callable` subdirectory of the `external` subdirectory of the main IDL directory. Note that different UNIX systems have different compilation and link statements. Note also that the name of the entry point in the object may be different than that shown here, because compilers may add leading or trailing underscores to the name of the source routine.

Note

The `Makefile` in the architecture-specific subdirectory of the `bin` subdirectory of the IDL distribution contains a make rule for building the `calltest` application.

Issues and Examples: Microsoft Windows

Building an Application that Calls IDL

To build your 32-bit, Win32 application that calls IDL, you must take the following steps:

1. Use a #include line to include the declarations from `idl_export.h` into your source code. This include file is found in the `external/include` subdirectory of the IDL distribution.
2. Compile your application.
3. Link your application with `IDL32.LIB`.
4. Place `IDL32.DLL` in a directory with your application. See the `readme.txt` file located in the *`RSI-directory/`*`external/callable` for more information.

Example: A Simple Application

The following program demonstrates how to display message text sent from IDL, execute IDL statements entered by a user, and how to obtain data from IDL variables. It performs the following actions:

1. Creates a Main window with four client controls; a scrolling edit control to display text messages from IDL, a single line edit control to allow a user to enter an IDL command, a **Send** button to send the user command to IDL, and a **Quit** button to exit the application.
2. Registers a callback function to handle text messages sent by IDL to the application.
3. Initializes Callable IDL.
4. Call **IDL_Cleanup()** when we receive the **WM_CLOSE** message.

Each line is numbered to make discussion easier. These numbers are not part of the actual program. The source code for this program can be found in the file `simple.c`, located in the `callable` subdirectory of the `external` subdirectory of the IDL distribution. See the source code for details of the program not printed here.

```
/*-----------------------------------------------------------
* simple.c Source code for sample IDL callable application
*
* Copyright (c) 1992-1995, Research Systems Inc.
*-----------------------------------------------------------
*/
#include <windows.h>
#include <windowsx.h>
#include <ctl3d.h>
#include <string.h>
#include <stdio.h>
#include "simple.h"
#include "idl_export.h"

/*-----------------------------------------------------------
 * WinMain
 *
 * This is the required entry point for all windows
applications.
 *
 * RETURNS:      TRUE if successful
 *---------------------------------------------------------*/
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hInstancePrev,
    LPSTR lpszCmndline, int nCmdShow)
{
    HWND              hwnd;
    MSG               msg;

    // Register the main window class.
    if (!RegisterWinClass(hInstance)) {
        return(0);
    }

        ...

    // Create and display the main window.
    if ((hwnd = InitMainWindow(hInstance)) == NULL) {
        return(0);
    }
    MainhWnd = hwnd;

    // Register our output function with IDL.
    IDL_ToutPush(OutFunc);

    // Initialize IDL
    if (!IDL_Win32Init(0, hInstance, hwnd, NULL))
        return(FALSE);

    // Main message loop.
    while (GetMessage(&msg, NULL, 0, 0)) {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }

    return(msg.wParam);
}

/*-------------------------------------------------------------
 * RegisterWinClass
 *
 * To create a Main window (TLB in IDL speak). You must first
 * register the class for that window
 *
 * RETURNS:      TRUE if successful
 *-----------------------------------------------------------*/
BOOL RegisterWinClass(HINSTANCE hInst)
{
    WNDCLASS            wc;

    wc.style            = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc      = MainWndProc;
    wc.cbClsExtra       = 0;
    wc.cbWndExtra       = 0;
    wc.hInstance        = hInst;
    wc.hIcon            = NULL;
    wc.hCursor          = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground    = (HBRUSH)(COLOR_BTNFACE + 1);
    wc.lpszMenuName     = NULL;
    wc.lpszClassName    = "Simple";

    if (!RegisterClass(&wc)) {
        return(FALSE);
    }

    return(TRUE);
}

/*-------------------------------------------------------------
 * InitMainWindow
 *
 * This is where our Main window is created and displayed
 *
 * RETURNS:      Handle to window
 *-----------------------------------------------------------*/
HWND InitMainWindow(HINSTANCE hInst)
{
    HWND                hwnd;
```

```
   CREATESTRUCT        cs;


   hwnd = CreateWindow("Simple",
       "Callable IDL Sample Application",
       WS_DLGFRAME | WS_SYSMENU | WS_MINIMIZEBOX | WS_VISIBLE,
       CW_USEDEFAULT,
       0,
       600,
       480,
       NULL,
       NULL,
       hInst,
       &cs);

   if (hwnd) {
      ShowWindow(hwnd, SW_SHOWNORMAL);
      UpdateWindow(hwnd);
   }

   return(hwnd);
}

/*-------------------------------------------------------------
 * MainWndProc
 *
 * The window procedure (event handler) for our main window.
 * All messages (events) sent to our app are routed through
 * here
 * RETURNS:          Depends of message.
 *-----------------------------------------------------------*/
LRESULT WINAPI MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
   static int               nDisplayable = 0;


   switch (uMsg) {
   //When our app is first created, we are sent this message.
   //We take this opportunity to create our child controls and
  //place them in their desired locations on the window.
      case WM_CREATE:
       if (!CreateControls(((LPCREATESTRUCT)lParam)->hInstance, hwnd)) {
           return(0);
       }
       if (!LayoutControls(hwnd)) {
           return(0);
       }
       nDisplayable = GetCharacterHeight(GetDlgItem(hwnd, IDE_COMMANDLOG));
       break;
```

```

         ...

        case WM_DESTROY:
            PostQuitMessage(1);
            break;

      //Each time a button or menu item is selected, we get this message
        case WM_COMMAND:
            OnCommand(hwnd, LOWORD(wParam), wParam, lParam);
            return(FALSE);

    //This is a message we send ourselves to indicate the need to
   //display a text message in our log window.
        case IDL_OUTPUT:
            OutputMessage(wParam, lParam, nDisplayable);
            return(FALSE);

        case WM_CLOSE:
            IDL_Cleanup(TRUE);
            return(FALSE);

        default:
            break;
     }

     return(DefWindowProc(hwnd, uMsg, wParam, lParam));
}

/*--------------------------------------------------------------
 * OnCommand
 *
 * This is the message handle for our WM_COMMAND messages
 *
 * RETURNS:           FALSE
 *-----------------------------------------------------------*/
BOOL OnCommand(HWND hWnd, UINT uId, WPARAM wParam, LPARAM lParam)
{

     switch(uId){
        case IDB_SENDCOMMAND:{
            LPSTR     lpCommand;
            LPSTR     lpOut;

            lpCommand = GlobalAllocPtr(GHND, 256);
            lpOut = GlobalAllocPtr(GHND, 256);
            if(!lpCommand)
               return(FALSE);

        /* First we get the string that is in the input window */
            GetDlgItemText(hWnd, IDE_COMMANDLINE, lpCommand,
255);

            /* and then clear the window */
            SetDlgItemText(hWnd, IDE_COMMANDLINE, "");

            lstrcpy(lpOut, "\r\nSent to IDL: ");
            lstrcat(lpOut, lpCommand);

            /* Send the string to our "log" window */
            OutFunc(IDL_TOUT_F_NLPOST, lpOut, strlen(lpOut));

            /* then send the string to IDL */
            IDL_ExecuteStr(lpCommand);

            /* Now clean up */
            GlobalFreePtr(lpCommand);
            GlobalFreePtr(lpOut);
      }
            break;
 }
 return(FALSE);
}

/*-------------------------------------------------------------
 * OutFunc
 *
 * This is the output function that receives messages from IDL
 * and displays them for the user
 *
 * RETURNS:          NONE
 *-----------------------------------------------------------*/
void OutFunc(long flags, char *buf, long n)
{
     static    fShowMain = FALSE;

     /* If there is a message, post it to our MAIN window */
     if (n){
         SendMessage (MainhWnd, IDL_OUTPUT, 0, (LPARAM)buf);
     }

     /* If we need to post a new line message... */
     if (flags & IDL_TOUT_F_NLPOST){
         SendMessage (MainhWnd, IDL_OUTPUT, 0, (LPARAM)(LPSTR)"\r\n\0");
     }

 /* This message gets sent to the log window to have it scroll
      and display the last message at the bottom of the window.
    With this, the user will always see the last screen full of
   messages sent
   */
   SendMessage (MainhWnd, IDL_OUTPUT, (WPARAM)TRUE,
                (LPARAM)(LPSTR)"\0");

   return;
}

/*-------------------------------------------------------------
 * OutputMessage
 *
 * Here we do the actual display of the text to our log window
 *
 * RETURNS:          nothing
 *
 *----------------------------------------------------------*/
void OutputMessage(WPARAM wParam, LPARAM lParam, int nDisplayable)
{
   LRESULT   lRet;
   LONG      lBufflen, lNumLines, lFirstView;

   /* Turn off the READONLY bit and postpone redraw */
   lRet = SendMessage(hwndLog, EM_SETREADONLY, FALSE, 0L);
   lRet = SendMessage(hwndLog, WM_SETREDRAW, FALSE, 0L);

  /* Get the length of the text in the log window*/
  lBufflen = SendMessage (hwndLog, WM_GETTEXTLENGTH, 0, 0L);
  lNumLines = SendMessage (hwndLog, EM_GETLINECOUNT, 0, 0L);
  lFirstView = SendMessage (hwndLog, EM_GETFIRSTVISIBLELINE, 0, 0L);
  lRet = SendMessage (hwndLog, EM_SETSEL, lBufflen, lBufflen);

   /* If we are adding text, wParam will be 0 */
 if(!wParam)
    lRet = SendMessage (hwndLog, EM_REPLACESEL, 0, lParam);
 else{
    if (lNumLines > (lFirstView + nDisplayable)){
         int        iLineLen = 0;
         int        iChar;
         int        iLines = 0;
         lNumLines--;
         while(!iLineLen){
             iChar = SendMessage(hwndLog, EM_LINEINDEX,
                     (WPARAM)lNumLines, 0L);
             iLineLen = SendMessage(hwndLog, EM_LINELENGTH,
                     iChar, 0L);
             if(!iLineLen)
                 lNumLines--;
         }
295          iLines = lNumLines-(lFirstView + (nDisplayable - 1));
296         iLines = iLines >= 0 ? iLines : 0;
297         SendMessage (hwndLog, EM_LINESCROLL, 0, (LPARAM)iLines);
298     }
299  }
300
301    /* Set the window to redraw and reset the READONLY bit */
302    lRet = SendMessage(hwndLog, WM_SETREDRAW, TRUE, 0L);
303    lRet = SendMessage(hwndLog, EM_SETREADONLY, TRUE, 0L);
304
305    return;
306 }
```

The following is a commentary on the program, by line number:

16

`idl_export.h` contains the **IDL_** function prototypes, IDL specific structures, and IDL constants.

45

Call **IDL_ToutPush()** with the address of the output function (**OutFunc**) as it's only argument. This will register **OutFunc** as a callback for IDL. IDL will call **OutFunc** when it needs to display text.

48

Initialize IDL with the handle to the main window and the HINSTANCE of the application.

52

Start the windows message loop.

131-176

This is the Main window procedure. It will handle any messages that are sent to the main window. This includes **WM_COMMAND** messages that occur as a result of user interaction with the client controls. In addition, it handles a user defined message called **IDL_OUTPUT** (the name doesn't matter but this is a clue as to its purpose).

158

When the user presses either the "Send" or "Quit" buttons, route the message to the **OnCommand** function.

164

When we receive an **IDL_OUTPUT** message, call the function that displays text in the scrolling window (**OutputMessage**. See line 263).

168

When we receive the **WM_CLOSE** message, call **IDL_Cleanup()** to unlink IDL from our application.

185-220

OnCommand handles the **WM_COMMAND** messages generated when the user clicks on the application's buttons.

199

Get the IDL command that the user has entered in the single line edit control and store it in a buffer.

202

Clear the text in the edit control.

208

Call the **IDL_TOUT_** function to display the command sent to IDL in the output window.

211

Call **IDL_ExecuteStr()** with the IDL command retrieved in line 199.

230-253

OutFunc is the callback registered with IDL to handle text messages IDL sends to our application. In addition it will handle text from IDL routines that display information, such as PRINT.

263-306

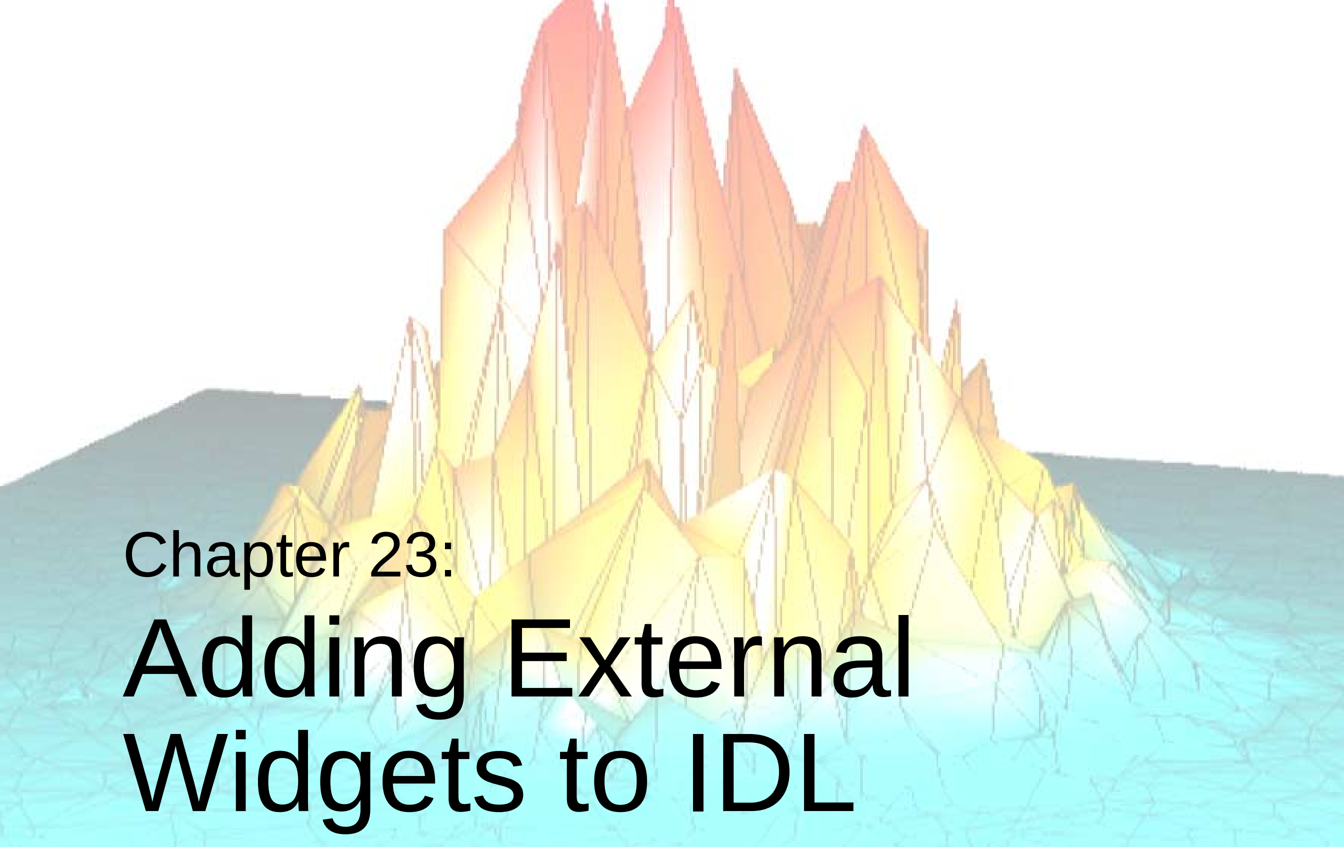
OutputMessage handles displaying the text to the output window. Since this window is a multi-line edit control, we have created it as a read-only window. See the source code for additional information on handling this situation.

280

OutputMessage appends new messages to the existing text in the control.

281-299

When the text has been displayed, **OutputMessage** scrolls the window to display the last line of text in the bottom of the window.



Chapter 23: Adding External Widgets to IDL

This chapter discusses the following topics:

IDL and External Widgets

This chapter describes an IDL widget type not documented in the *IDL Reference Guide*, called the *stub widget*. It also describes a small set of internal functions to manipulate stub widgets. Stub widgets allow CALL_EXTERNAL, LINKIMAGE, DLM, and Callable IDL users to add their own widgets to IDL widget hierarchies.

This feature depends on your system providing the window system libraries used by IDL (particularly the Motif libraries under UNIX) as sharable libraries. It will not work with versions of IDL that statically link against the window system libraries. This can be an issue under Linux, but one that we expect to eventually disappear as Linux distributions start shipping Open Motif as a standard part of the systems.

The next two sections describe IDL's WIDGET_STUB function and changes to WIDGET_CONTROL when used with WIDGET_STUB. "Functions for Use with Stub Widgets" on page 506 describes support functions that can be called from your external code to manipulate stub widgets. "Internal Callback Functions" on page 509 describes how to make stub widgets generate IDL widget events. Finally, "UNIX WIDGET_STUB Example: WIDGET_ARROWB" on page 511 illustrates the use of stub widgets with an external program.

Note

Although WIDGET_STUB can be used under Microsoft Windows, this feature is primarily of interest with UNIX IDL. Under Windows, RSI recommends the use of the WIDGET_ACTIVEX functionality, which allows you to use ActiveX controls with IDL without requiring external programming.

WIDGET_STUB

The WIDGET_STUB function creates a widget record that contains no actual underlying widgets. Stub widgets are place holders for integrating external widget types into IDL. Events from those widgets can then be processed in a manner consistent with the rest of the IDL widget system.

First, the programmer calls WIDGET_STUB to create the widget, and then uses CALL_EXTERNAL to call additional custom code to handle the rest. A number of internal functions are provided to manipulate widgets from this custom code. See "Functions for Use with Stub Widgets" on page 506.

The returned value of this function is the widget ID of the newly-created stub widget.

Calling Sequence

Result = WIDGET_STUB(*Parent*)

Arguments

Parent

The widget ID of the parent widget. Stub widgets can only have bases or other stub widgets as their parents.

Keywords

The following keywords are accepted by WIDGET_STUB and work the same as for other widget creation functions:

EVENT_FUNC	SCR_XSIZE
EVENT_PRO	SCR_YSIZE
FUNC_GET_VALUE	UVALUE
GROUP_LEADER	XOFFSET
KILL_NOTIFY	XSIZE
NO_COPY	YOFFSET
PRO_SET_VALUE	YSIZE

WIDGET_CONTROL/WIDGET_STUB

The WIDGET_CONTROL procedure has some differences and limitations when used with WIDGET_STUB that are not documented in the *IDL Reference Guide*. These differences are described below.

Keywords

Only the most general keywords are allowed with WIDGET_CONTROL when used with stub widgets. All other keywords are ignored. Here is a list of those keywords that behave identically with all widgets including stub widgets:

BAD_ID	PRO_SET_VALUE
CLEAR_EVENTS	RESET
EVENT_FUNC	SET_UVALUE
EVENT_PRO	SHOW
FUNC_GET_VALUE	TIMER
GET_UVALUE	TLB_GET_OFFSET
GROUP_LEADER	TLB_GET_SIZE
HOURGLASS	TLB_SET_TITLE
ICONIFY	TLB_SET_XOFFSET
KILL_NOTIFY	TLB_SET_YOFFSET
MANAGED	XOFFSET
NO_COPY	YOFFSET

The following keywords also work with stub widgets, but require additional commentary:

DESTROY

When a widget hierarchy containing stub widgets is destroyed, the following steps are taken:

- The lower-level code that deals with the system toolkit destroys any real widgets currently used by the stub widgets.
- All IDL widget records are added to the free list for re-use.

- Any requested KILL_NOTIFY callbacks are called.

You should register KILL_NOTIFY callbacks on the topmost stub widget in each widget subtree. Remember that the actual widgets are gone before the callbacks are issued, so don't attempt to access them. However, the callback provides an opportunity to clean up any related resources used by the widget.

MAP, REALIZE, and SENSITIVE

These keywords cause the toolkit-specific, lower layer of the IDL widgets implementation to be called. In the process of satisfying the specified request, any real widgets used by the stub widgets will be processed, along with the ones created by the non-stub widgets, in the usual way. Any additional processing must be provided via CALL_EXTERNAL.

XSIZE, SCR_XSIZE, YSIZE, and SCR_YSIZE

These keywords inform IDL how large the stub widget is expected to be. This information is necessary for IDL to calculate sizes and offsets of the surrounding widgets.

IDL tries to do something reasonable with these requests but, without knowledge of the actual widget being manipulated, it is possible that the results will not be satisfactory. In such cases, the **IDL_WidgetStubSetSizeFunc()** function can be used to specify a routine that IDL can call to perform the necessary sizing for your stub widget.

Functions for Use with Stub Widgets

The following functions present a highly simplified interface to the stub widget class that gives the user enough access to IDL widget internals to make the stub widget work while hiding the details of the actual implementation.

IDL_WidgetStubLock()

Syntax:

```
void IDL_WidgetStubLock(int set);
```

IDL event processing occurs asynchronously, so any code that manipulates widgets *must* execute in a protected region. This function is used to create such a region. Any code that manipulates widgets must be surrounded by two calls to **IDL_WidgetStubLock()** as follows:

```
IDL_WidgetStubLock(TRUE);
   /* Do your widget stuff */
IDL_WidgetStubLock(FALSE);
```

IDL_WidgetStubLookup()

Syntax:

```
char *IDL_WidgetStubLookup(IDL_ULONG id);
```

When IDL creates a widget, it returns an integer value to the caller of the widget creation function. Internally, however, IDL widgets are represented by a pointer to memory. The **IDL_WidgetStubLookup()** function is used to translate the user-level integer value to this memory pointer. All the other internal routines use the memory pointer to reference the widget.

Id is the integer returned at the user level. Your call to CALL_EXTERNAL should pass this integer to your C-level code for use with **IDL_WidgetStubLookup()** which translates the integer to the pointer.

If the specified id does not represent a valid IDL widget, this function returns NULL. This situation can occur if a widget was killed but its integer handle is still lingering somewhere.

IDL_WidgetIssueStubEvent()

Syntax:

```
void IDL_WidgetIssueStubEvent(char *rec, LONG value);
```

Given a handle to the IDL widget, obtained via **IDL_WidgetStubLookup()**, this function queues an IDL **WIDGET_STUB_EVENT**. Such an event is a structure that contains the three standard fields (ID, TOP, and HANDLER) as well as an additional field named VALUE that contains the specified **value**.

VALUE can provide a way to access additional information about the widget, possibly by providing a memory address to the information.

IDL_WidgetSetStubIds()

Syntax:

```
void IDL_WidgetSetStubIds(char *rec, unsigned long t_id,
                            unsigned long b_id);
```

IDL widgets are built out of one or more actual widgets. Every IDL widget carries two pointers that are used to locate the top and bottom real widget for a given IDL widget. This function allows you to set these top and bottom pointers in the stub widget for later use.

Since the actual pointer type differs from toolkit to toolkit, this function declares **t_id** (the top real widget) and **b_id** (the bottom real widget) as unsigned long, an integer data type large enough to safely contain any pointer. Use a C cast operator to handle the difference.

After calling WIDGET_STUB to create an IDL stub widget, you will need to use CALL_EXTERNAL to call additional code that creates the real widgets that represent the stub. Having done that, use **IDL_WidgetSetStubIds()** to save the top and bottom widget pointers.

IDL_WidgetGetStubIds()

Syntax:

```
void IDL_WidgetGetStubIds(char *rec, unsigned long *t_id,
                            unsigned long *b_id);
```

This function returns the top (**t_id**) and bottom (**b_id**) real widget pointers for any specified widget (not just stub widgets). When using these values for non-stub widgets, it is the caller’s responsibility to avoid damaging the IDL-created widgets in any way.

IDL_WidgetStubSetSizeFunc()

Syntax:

```
void IDL_WidgetStubSetSizeFunc(char *rec,
                               IDL_WIDGET_STUB_SET_SIZE_FUNC func)

typedef void (* IDL_WIDGET_STUB_SET_SIZE_FUNC);
              (IDL_ULONG id, int width, int height);
```

When IDL needs to set the size of a stub widget, it attempts to set the size of the bottom real widget to the necessary dimensions. Often, this is the desired behavior, but cases can arise where it would be better to handle sizing differently. In such cases, use **IDL_WidgetStubSetSizeFunc()** to register a function that IDL will call to do the actual sizing.

Internal Callback Functions

Real widget toolkits (upon which IDL widgets are built) are event driven. C language programs register interest in specific events by providing callback functions that are called when that event occurs. All but the most basic of widgets are capable of generating events.

In order for IDL stub widgets to generate IDL events, you must use CALL_EXTERNAL to invoke code that sets up real widget event callbacks for the events you are interested in. This setup can be done as part of creating the real widgets after the initial call to WIDGET_STUB. These callbacks then call **IDL_WidgetIssueStubEvent()** to issue the IDL event.

Your C-language widget toolkit callback functions should be patterned after the following template. Note that the arguments and return type will depend on the widget toolkit used, and so cannot be shown here:

```
stub_widget_call()
{
  char *idl_widget;
  IDL_WidgetStubLock(TRUE);
    /* Get the IDL user-level identifier for this widget */
    if (idl_widget = IDL_WidgetStubLookup(id)) {
       /* Do whatever work is required */
        ...
       /* Optionally, issue an IDL event */
       IDL_WidgetIssueStubEvent(idl_widget, value)
    }
  IDL_WidgetStubLock(FALSE);
}
```

Commentary on the Example Shown Above

Note that **IDL_WidgetStubLock()** is used to protect the critical section where widgets are being manipulated.

Somehow, the callback must be able to find the user-level integer returned by WIDGET_STUB when the stub widget was created in IDL. Usually, this is done in one of two ways:

- When registering the callback, it is sometimes possible to specify a value that will be passed to the callback without interpretation. For example, the X windows **XtAddCallback()** function takes an argument named **client_data**. This value is passed to the callback and can be used to supply the user-level identifier.

- Some widget toolkits have a set of attributes that they carry along with each widget. Under the X windows Xt toolkit, these attributes are called resources. Xt widgets usually have a resource capable of holding a single integer or memory address. This resource can be used to supply the user level identifier.

IDL_WidgetStubLookup() is used to translate the user level widget identifier into a memory pointer. If this function returns NULL, no further event processing is done since it would be a fatal error to issue an IDL event for a non-existent widget.

The event is issued via **IDL_WidgetIssueStubEvent()**. This step is not required. Many of the IDL widget types process real widget events via callbacks that do not always result in an IDL widget event being sent.

UNIX WIDGET_STUB Example: WIDGET_ARROWB

The following example adds the Motif ArrowButton widget to UNIX IDL in the form of an IDL program named `widget_arrowb.pro`.

The primary user interface to our arrow button widget is the WIDGET_ARROWB function. It presents an interface much like any of the built in WIDGET_ functions provided by IDL. WIDGET_ARROWB uses the MAKE_DLL procedure, and the AUTO_GLUE keyword to CALL_EXTERNAL to automatically build and load the C code required for this widget. This building and loading process is transparent to the IDL user, requiring only that you have a C compiler installed on your system. All the user has to do to use an arrow button widget is to call WIDGET_ARROWB

The WIDGET_ARROWB widget acts like a normal pushbutton. Events are sent when the button is pressed (VALUE=1) and released (VALUE=0). If the USE_OWN_SIZE keyword is set to zero, IDL performs its default sizing on the stub widget. A non-zero value causes a special routine provided by the WIDGET_ARROWB implementation to be registered to handle such sizing.

All of the code used in this example, including all code shown here, is available in the external/widstub directory of the IDL distribution. To run it, execute the following statements from IDL:

```
PUSHD, FILEPATH('', SUBDIRECTORY=['external','widstub'])
WIDGET_ARROWB_TEST
POPD
```

When running WIDGET_ARROWB_TEST, you can specify the VERBOSE keyword, in which case, it will show you the compilation and linking steps it takes to build the sharable library from the C code. The use of pushd and popd are due to the fact that your IDL search path (!PATH) is unlikely to have the directory containing these examples in it. PUSHD changes your working directory to the location where these files are found, and POPD restores it to its original location afterwards.

The IDL Program for WIDGET_ARROWB

The following text is the IDL program for WIDGET_ARROWB. It is found in the file named `WIDGET_ARROWB.PRO`:

```
function WIDGET_ARROWB, parent, use_own_size, UVALUE=uvalue, $
                        VERBOSE=verbose, _EXTRA=extra
  ; Uses WIDGET_STUB, and a sharable library containing
  ; the necessary C support code, to provide the IDL user
  ; with a Motif Arrow Button widget. The interface is consistent
  ; with that presented by the built in IDL widgets.
  ;
  ; If the sharable library does not exist, it is built using
  ; MAKE_DLL.

  common WIDGET_ARROWB_BLK, shlib

  ; Build sharable lib if first call or lib doesn’t exist
  build_lib = n_elements(shlib) eq 0
  if (not build_lib) then build_lib = not FILE_TEST(shlib, /READ)
  if (build_lib) then begin
    ; Location of the widget_arrowb files from IDL distribution
    arrowb_dir=FILEPATH('',SUBDIRECTORY=['external','widstub'])

    ; Use MAKE_DLL to build the widget_arrowb sharable library
    ; in the !MAKE_DLL.COMPILE_DIRECTORY directory.
    ;
    ; Normally, you wouldn’t use VERBOSE, or SHOW_ALL_OUTPUT
    ; once your work is debugged, but as a learning exercize it
    ; can be useful to see all the underlying work that gets
    ; done. If the user specified VERBOSE, then use those
    ; keywords to show what MAKE_DLL is doing.
    MAKE_DLL,'widget_arrowb', 'widget_arrowb', $
             DLL_PATH=shlib, INPUT_DIR=arrowb_dir, $
             VERBOSE=verbose,SHOW_ALL_OUTPUT=verbose
  endif

  ; Use a stub widget along with the C code in the library to
  ; create an arrow button widget. The use of the AUTO_GLUE
  ; keyword simplifies the call to the sharable library by
  ; eliminating the need to use the CALL_EXTERNAL portable
  ; calling convention.
  l_parent=LONG(parent)
  l_use_own_size = $
      (n_elements(use_own_size) eq 0) ? 0L: LONG(use_own_size)
  result = WIDGET_STUB(parent, _extra=extra)
  if (n_elements(uvalue) ne 0) then $
    WIDGET_CONTROL, result, set_uvalue=uvalue
  JUNK = CALL_EXTERNAL(shlib, 'widget_arrowb',l_parent,result,$
                l_use_own_size, value=[1, 1, 1], /AUTO_GLUE)

  RETURN, result
end
```

The C Program for widget_arrowb.c

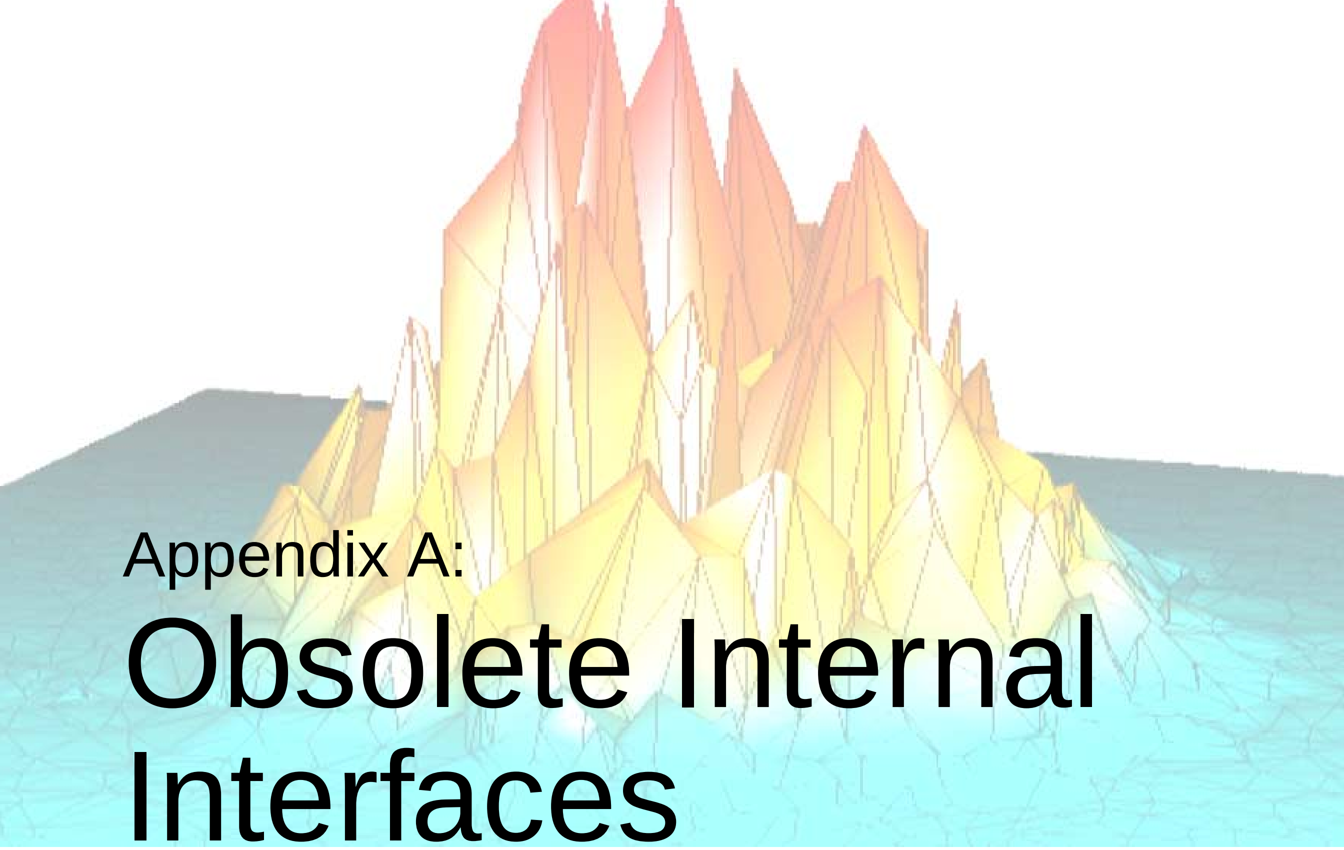
The C language code invoked by the call to CALL_EXTERNAL in the above IDL code is contained in a file named `widget_arrowb.c` This file can be found in the `widstub` subdirectory of the `external` subdirectory of the IDL distribution. The contents of this file are shown below:

```
/*
 * widget_arrowb.c - This file contains C code to be called from
 * UNIX IDL via CALL_EXTERNAL. It uses the IDL stub widget to add
 * a Motif ArrowButton to an IDL created widget hierarchy. The
 * button issues a WIDGET_STUB_EVENT every time the button is
 * released.
 *
 * While this code is Motif-centric, the principles apply across  *
platforms and could be adapted to Microsoft Windows.
 */
#include <stdio.h>
#include <X11/keysym.h> /* Keysyms for text widget events */
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Shell.h>
#include <Xm/ArrowB.h>
#include "idl_export.h"

/*ARGSUSED*/
static void arrowb_CB(Widget w, caddr_t client_data,
                      caddr_t call_data)
{
  char *rec;
  XmArrowButtonCallbackStruct *abcs;

  IDL_WidgetStubLock(TRUE);
if (rec = IDL_WidgetStubLookup((unsigned long) client_data)) {
    abcs = (XmArrowButtonCallbackStruct *) call_data;
    IDL_WidgetIssueStubEvent(rec, abcs->reason == XmCR_ARM);
  }
  IDL_WidgetStubLock(FALSE);
}


static void arrowb_size_func(IDL_ULONG stub, int width,
                             int height)
{
  char *stub_rec;
  unsigned long t_id, b_id;
  char buf[128];
  IDL_WidgetStubLock(TRUE);
  if (stub_rec = IDL_WidgetStubLookup(stub)) {
    IDL_WidgetGetStubIds(stub_rec, &t_id, &b_id);
    sprintf(buf, "Setting WIDGET %d to width %d and height %d",
            stub, width, height);
    IDL_Message(IDL_M_NAMED_GENERIC, IDL_MSG_INFO, buf);
    XtVaSetValues((Widget) b_id, XmNwidth, width, XmNheight,
                  height, NULL);
  }
  IDL_WidgetStubLock(FALSE);
}
int widget_arrowb(IDL_LONG parent, IDL_LONG stub, IDL_LONG
                  use_own_size_func)
{
  Widget parent_w;
  Widget stub_w;
  char *parent_rec;
  char *stub_rec;
  unsigned long t_id, b_id;

  IDL_WidgetStubLock(TRUE);
  if ((parent_rec = IDL_WidgetStubLookup(parent))
      && (stub_rec = IDL_WidgetStubLookup(stub))) {
    /* Bottom widget of parent is parent to arrow button */
    IDL_WidgetGetStubIds(parent_rec, &t_id, &b_id);
    parent_w = (Widget) b_id;
    stub_w = XtVaCreateManagedWidget("arrowb",
                                     xmArrowButtonWidgetClass,
                                     parent_w, NULL);
    IDL_WidgetSetStubIds(stub_rec, (unsigned long) stub_w,
                         (unsigned long) stub_w);
    XtAddCallback(stub_w, XmNarmCallback,
                  (XtCallbackProc) arrowb_CB, (XtPointer) stub);
    XtAddCallback(stub_w, XmNdisarmCallback,
                  (XtCallbackProc) arrowb_CB, (XtPointer) stub);
    if (use_own_size_func)
      IDL_WidgetStubSetSizeFunc(stub_rec, arrowb_size_func);
  }
  IDL_WidgetStubLock(FALSE);
  return stub;
}
```

An IDL Program to Test the External Widget

Shown below is an IDL widget program to test the ARROWB widget. This program is found in the file widget_arrowb_test.pro in the IDL distribution:

```
pro widget_arrowb_test_event, ev
  widget_control, get_uvalue=val, ev.id
  if (val eq 0) then begin
    widget_control, /destroy, ev.top
  endif else begin
    HELP, /STRUCT, ev
    if (ev.value eq 1) then begin
      widget_control,val,set_value=’New label string’
      tmp = widget_info(ev.id,/GEOMETRY)
      widget_control, xsize=tmp.xsize+25, $
                      ysize=tmp.ysize+25, ev.id
    endif
  endelse
end

pro widget_arrowb_test, VERBOSE=verbose
  a = widget_base(/COLUMN)
  b = widget_button(a, value=’Done’, uvalue = 0)
  label=widget_label(a,value=’A label’)
  arrow_w = widget_arrowb(a, 0, xsize=100, ysize=100, $
                          uvalue=label, verbose=verbose)
  arrow_w = widget_arrowb(a, 1, xsize=100, ysize=50, $
                          uvalue=label, verbose=verbose)
  widget_control,/real,a
  xmanager, ’WIDGET_ARROWB_TEST’, a, /NO_BLOCK
end
```



Appendix A:

Obsolete Internal Interfaces

This chapter discusses the following topics:

Interfaces Obsoleted in IDL 5.5

The following areas changed in IDL 5.5, requiring the introduction of new interfaces, and causing some old interfaces to become obsolete. These old interfaces remain in IDL and can be used by user code. However, new code should not use them, and old code might benefit from migration as part of normal maintenance:

- The **IDL_Message**() **IDL_MSG_ATTR_SYS** attribute has been retired, in favor of the more general **IDL_MessageSyscode**() function.
- The **IDL_MessageErrno**() and **IDL_MessageErrnoFromBlock**() functions have been retired in favor of the **IDL_MessageSyscode**() and **IDL_MessageSyscodeFromBlock**() functions, which are more general.
- IDL's keyword API has been redesigned to be easier to use and understand, and to be reentrant.

IDL_MSG_ATTR_SYS

Note

IDL_MSG_ATTR_SYS is one of the possible attribute values that can be included in the **action** argument to the **IDL_Message**() function. Its purpose was to cause **IDL_Message**() to report the system error currently contained in the process **errno** global variable. This functionality is now available in a more general and useful form via the **IDL_MessageSyscode**() and **IDL_MessageSyscodeFromBlock**() functions, documented in "Issuing Error Messages" on page 338

IDL_MSG_ATTR_SYS

I**IDL_Message()** always issues a single-line error message that describes the problem from IDL's point of view. Often, however, there is an underlying system reason for the error that should also be displayed to give the user a complete picture of what went wrong. For example, the IDL view of the problem might be "Unable to open file", while the underlying system reason for the error is "no such directory".

The UNIX system provides a global variable named **errno** for communicating such system level errors. Whenever a call to a system function fails, it returns a 1, and puts an error code into **errno** that specifies the reason for the failure. Other functions, such as those provided by the standard C library, do not set **errno**. These functions do set **errno**.

Specifying **IDL_MSG_ATTR_SYS** tells **IDL_Message()** to check **errno**, and if it is non-null, to issue a second line containing the text of the system error message.

Specify **IDL_MSG_ATTR_SYS** only if you are calling **IDL_Message()** as the result of a failed UNIX system call. Otherwise, **errno** might contain an unrelated garbage value resulting in an incorrect error message.

The Microsoft Windows operating system has **errno** for compatibility with the expectations of C programmers, but typically do not set it. On these operating systems, it is possible to specify **IDL_MSG_ATTR_SYS**, but it has no effect.

Specifying errno Explicitly: IDL_MessageErrno()

Note

The **IDL_MessageErrno**() and **IDL_MessageErrnoFromBlock**() functions allow you to throw an error message that includes the system error from the UNIX/POSIX **errno** global variable. These functions have been replaced by **IDL_MessageSyscode**() and **IDL_MessageSyscodeFromBlock**() which in addition to being able to throw UNIX/Posix errors, can also throw other types of system error.

There are times when specifying the **IDL_MSG_ATTR_SYS** modifier code in the action argument to **IDL_Message()** is inadequate. This situation usually occurs when your code attempts to perform some cleanup operation when an operating system call fails before calling **IDL_Message()** and this cleanup code might alter the value of **errno**. In such cases, it is preferable to use the **IDL_MessageErrno()** or **IDL_MessageErrnoFromBlock()** functions to issue the message:

```
void IDL_MessageErrno(int code, int errno, int action, …)
void IDL_MessageErrnoFromBlock(IDL_MSG_BLOCK block, int code, int
errno, int action, ...)
```

These function differs from **IDL_Message()** in two ways:

1. There is an additional argument used to specify the value of **errno**. See the discussion of **errno** in “IDL_MSG_ATTR_SYS” on page 518 for additional information about **errno** and its use.
2. The **IDL_MSG_ATTR_SYS** modifier code for the action argument is ignored.-

Processing Keywords With IDL_KWGetParams()

Note

Previous versions of IDL used a keyword API based around the **IDL_KWGetParams()** and **IDL_KWCleanup()** functions. This API was confusing to use (It was difficult to know when **IDL_KWCleanup()** was supposed to be called), and was not reentrant (requiring extensive and error prone code in some IDL system routines) . The new API, using **IDL_KWProcessByOffset()** and IDL_KW_FREE, solve these problems and result in easier to write and maintain code.

To enable rapid conversion from the old API to the new, the new API uses most of the same data structures as the old (with the notable exception of IDL_KW_ARR_DESC, which is replaced by IDL_KW_ARR_DESC_R).

This section reproduces those parts of the documentation of the original API that differ from the current API, which is described in Chapter 13, “IDL Internals: Keyword Processing”

The IDL_KW_PAR Structure

Note

IDL_KW_PAR is used with the old keyword API in largely the same manner as the current API, as described in “Overview Of IDL Keyword Processing” on page 300. The main difference is that the contents of the **specified** and **value** fields are the addresses of static variables, rather than offsets into a **KW_RESULT** structure as with the new API.

specified

The address of a C int variable that will be set to TRUE (non-zero) or FALSE (0) based on whether the routine was called with the keyword present. This field should be set to NULL (**(int *) 0**) if this information is not needed.

value

If the keyword is a read-only scalar, this field is a pointer to a C variable of the correct type (IDL_LONG, IDL_ULONG, IDL_LONG64, IDL_ULONG64, float, double, or IDL_STRING).

In the case of a read-only array, value is a pointer to an **IDL_KW_ARR_DESC**, which is discussed in "The IDL_KW_ARR_DESC Structure" on page 521. In the case of an output variable (i.e., the **IDL_KW_OUT** flag is set), this field should point to an **IDL_VPTR** that will be filled by **IDL_KWGetParams()** with the address of the keyword argument.

The IDL_KW_ARR_DESC Structure

Note

The **IDL_KW_ARR_DESC** structure was superseded by **IDL_KW_ARR_DESC_R** in the current API. The reason for this change is that the **n** field of **IDL_KW_ARR_DESC** is modified by the call to **IDL_KWGetParams**(), requiring the **IDL_KW_ARR_DESC** structure to be defined in static memory, and rendering it non-reentrant.

When a keyword is specified to be a read-only array (i.e., the **IDL_KW_ARRAY** flag is set), the value field of the **IDL_KW_PAR** struct should be set to point to an **IDL_KW_ARR_DESC** structure. This structure is defined as:

```
typedef struct {
  char *data;
  IDL_MEMINT nmin;
  IDL_MEMINT nmax;
  IDL_MEMINT n;
} IDL_KW_ARR_DESC;
```

where:

data

The address of a C array to receive the data. This array must be of the C type mapped into by the **type** field of the **IDL_KW_PAR** struct. For example, **IDL_TYP_LONG** maps into a C **IDL_LONG**. There must be **nmax** elements in the array.

nmin

The minimum number of elements allowed.

nmax

The maximum number of elements allowed.

n

The number of elements actually present. Unlike the other fields, this field is set by **IDL_KWGetParams()**.

Processing Keywords

The **IDL_KWGetParams()** function is used to process keywords. **IDL_KWGetParams()** performs the following actions on behalf of the calling system routine:

- Verify that the keywords passed to the routine are all allowed by the routine.
- Carry out the type checking and conversions required for each keyword.
- Find the positional (non-keyword) arguments that are scattered among the keyword arguments in **argv** and copy them in order into the **plain_args** array.
- Return the number of plain arguments copied into **plain_args**.

IDL_KWGetParams() has the form:

```
int IDL_KWGetParams(int argc, IDL_VPTR *argv,char *argk,
        IDL_KW_PAR *kw_list, IDL_VPTR plain_args[], int mask)
```

where:

argc

The number of arguments passed to the caller. This is the first parameter to all system routines.

argv

The array of **IDL_VPTR** to arguments that was passed to the caller. This is the second parameter to all system routines.

argk

The pointer to the keyword list that was passed to the caller. This is the third parameter to all system routines that accept keyword arguments.

kw_list

An array of **IDL_KW_PAR** structures (see“Overview Of IDL Keyword Processing” on page 300 , and “The IDL_KW_PAR Structure” on page 520) that specifies the acceptable keywords for this routine. This array is terminated by setting the keyword field of the final struct to NULL (**(char *) 0**).

plain_args

An array of **IDL_VPTR** into which the **IDL_VPTR**s of the positional arguments will be copied. This array must have enough elements to hold the maximum possible number of positional arguments, as defined in **IDL_SYSFUN_DEF2**. See "Registering Routines" on page 438.

mask

Mask enable. This variable is ANDed with the mask field of each **IDL_KW_PAR** struct in the array given by **kw_list**. If the result is non-zero, the keyword is accepted as a valid keyword for the called system routine. If the result is zero, the keyword is ignored.

Speeding Keyword Processing

As mentioned above, the **kw_list** argument to **IDL_KWGetParams()** is a null terminated list of **IDL_KW_PAR** structures. The time required to scan each item of the keyword array and zero the required fields (those fields specified, and value fields with IDL_KW_ZERO set), can become significant, especially when more than a few keyword array elements (e.g., 5 to 10 elements) are present.

To speed things up, specify **IDL_KW_FAST_SCAN** as the first keyword array element. If **IDL_KW_FAST_SCAN** is the first keyword array element, the keyword array is compiled by **IDL_KWGetParams()** into a more efficient form the first time it is used. Subsequent calls use this efficient version, greatly speeding keyword processing. Usage of **IDL_KW_FAST_SCAN** is optional, and is not worthwhile for small lists. For longer lists, however, the improvement in speed is noticeable. For example, the following list does not use fast scanning:

```
static IDL_KW_PAR  kw_pars[] = {
  { "DOUBLE", IDL_TYP_DOUBLE, 1, 0, &d_there, CHARA(d) },
  { "FLOAT", IDL_TYP_FLOAT, 1, IDL_KW_ZERO, 0, CHARA(f) },
  { NULL }
};
```

To use fast scanning, it would be written as:

```
static IDL_KW_PAR  kw_pars[] = {
  IDL_KW_FAST_SCAN,
  { "DOUBLE", IDL_TYP_DOUBLE, 1, 0, &d_there, CHARA(d) },
  {"FLOAT", IDL_TYP_FLOAT, 1, IDL_KW_ZERO, 0, CHARA(f) },
  { NULL }
};
```

Cleaning Up

The **IDL_KWCleanup()** function is necessary if the keywords allowed by a system routine include any input-only keywords of type **IDL_TYP_STRING**, or if the **IDL_KW_VIN** flag is used by any of the keyword **IDL_KW_PAR** structures. Such keywords can cause keyword processing to allocate temporary variables that must be cleaned up after they've outlived their usefulness. Call **IDL_KWCleanup()** as follows:

```
void IDL_KWCleanup(int fcn)
```

where **fcn** specifies the operation to be performed, and must be one of the following values:

IDL_KW_MARK

Mark the stack by placing the statement:

```
IDL_KWCleanup(IDL_KW_MARK);
```

above the call to **IDL_KWGetParams()**. In addition, you will need to make a call with **IDL_KW_CLEAN** at the end.

IDL_KW_CLEAN

Clean up from the last call to **IDL_KWGetParams()** by placing the line:

```
IDL_KWCleanup(IDL_KW_CLEAN);
```

just above the **return** statement.

Keyword Examples

The following C function implements KEYWORD_DEMO, a system procedure intended to demonstrate how to write the keyword processing code for a routine. It prints the values of its keywords, changes the value of READWRITE to 42 if it is present, and returns. Each line is numbered to make discussion easier. These numbers are not part of the actual program.

Note

The following code is designed to demonstrate keyword processing in a simple, uncluttered example. In actual code, you would not use the **printf** mechanism used on lines 35-39.

```
        1 #include <stdio.h>
        2 #include <idl_export.h>
        3
        4 void keyword_demo(int argc, IDL_VPTR *argv, char *argk)
        5 {
        6   int i;
        7   IDL_ALLTYPES newval;
        8
        9   static int d_there, s_there, arr_there;
       10   static IDL_LONG l;
       11   static float f;
       12   static double d;
       13   static IDL_STRING s;
       14   static IDL_LONG arr_data[10];
C      15   static IDL_KW_ARRAY_DESC arr_d = {(char *) arr_data,3,10,0};
       16   static IDL_VPTR var;
       17
       18   static IDL_KW_PAR kw_pars[] = { IDL_KW_FAST_SCAN,
       19     { "ARRAY", IDL_TYP_LONG, 1, IDL_KW_ARRAY, &arr_there,
       20       IDL_CHARA(arr_d) },
       21     { "DOUBLE", IDL_TYP_DOUBLE, 1, 0, &d_there, IDL_CHARA(d) },
       22     { "FLOAT", IDL_TYP_FLOAT, 1, IDL_KW_ZERO, 0, IDL_CHARA(f) },
       23     { "LONG", IDL_TYP_LONG, 1, IDL_KW_ZERO|IDL_KW_VALUE|15, 0,
       24        IDL_CHARA(l) },
       25     { "READWRITE", IDL_TYP_UNDEF, 1, IDL_KW_OUT|IDL_KW_ZERO,
       26        0, IDL_CHARA(var) },
       27     { "STRING",TYP_STRING, 1, 0, &s_there, IDL_CHARA(s) },
       28     { NULL }
       29   };

IDL_KWCleanup(IDL_KW_MARK);

(void) IDL_KWGetParams(argc, argv, argk, kw_pars, NULL, 1);

printf("LONG: <%spresent>\n", l ? "": "not ");
printf("FLOAT: %f\n", f);
printf("DOUBLE: <%spresent>\n", d_there ? "": "not ");
printf("STRING: %s\n", s_there ? IDL_STRING_STR(&s) : "<not present>");
printf("ARRAY: ");
if (arr_there)
  for (i = 0; i < arr_d.n; i++)
    printf(" %d", arr_data[i]);
else
  printf("<not present>");
printf("\n");

printf("READWRITE: ");
if (var) {
  IDL_Print(1, &var, (char *) 0);
  newval.l = 42;
  IDL_StoreScalar(var, TYP_LONG, &newval);
} else {
  printf("<not present>");
}
printf("\n");

IDL_KWCleanup(IDL_KW_CLEAN);
}
```

Executing this routine from the IDL command line, by entering:

```
KEYWORD_DEMO
```

gives the output:

```
LONG: <not present>
FLOAT: 0.000000
DOUBLE: <not present>
STRING: <not present>
ARRAY: <not present>
READWRITE: <not present>
```

Executing it again with keywords specified:

```
A = 56
KEYWORD_DEMO, /LONG, FLOAT=2, DOUBLE=34,$
  STRING="hello", ARRAY=FINDGEN(10), READWRITE=A
```

```
PRINT, 'Final Value of A: ', A
```

gives the output:

```
LONG: <present>
FLOAT: 2.000000
DOUBLE: <present>
STRING: hello
ARRAY: 0 1 2 3 4 5 6 7 8 9
READWRITE:      56
Final Value of A:           42
```

Those features of this procedure that are interesting in terms of keyword processing are, by line number:

7

The **IDL_StoreScalar()** function used on line 51 requires the scalar to be provided in an **IDL_ALLTYPES** struct.

9

These variables are used to determine if a given keyword is present. Note that all the keyword-related variables are declared static. This is necessary so that the C compiler can build the **IDL_KW_PAR** structure at compile time.

10 – 13

C variables to receive the scalar read-only keyword values.

14

C array to be used for the ARRAY read-only array keyword.

15

The array descriptor used for ARRAY. **arr_data** is the address where the array contents should be copied. The minimum number of elements allowed is 3, the maximum is 10. The value set in the last field (0) is not important, because the keyword processing routine never reads its value. Instead, it puts the number of elements actually seen there.

16

The READWRITE keyword uses the **IDL_KW_OUT** flag, so the routine receives an **IDL_VPTR** instead of a processed value.

18

The keyword definition array. Notice that all of the keywords are ordered lexically (ASCII) and that there is a NULL entry at the end (line 28). Also, all of the mask fields are set to 1, as is the mask argument to **IDL_KWGetParams()** on line 33. This means that all of the keywords in the list are to be considered valid in this routine.

The IDL_KW_FAST_SCAN macro is used to define the first keyword array element, speeding the processing of a long IDL_KW_PAR list.

19 – 20

ARRAY is defined to be a read-only array keyword of IDL_TYP_LONG. The **arr_there** variable will be set to non-zero if the keyword is present. In that case, the array contents will be placed in the variable **arr_data** and the number of elements will be placed into **arr_d.n**.

21

DOUBLE is a scalar keyword of **IDL_TYP_DOUBLE**. It uses the variable **d_there** to know if the keyword is present.

22

FLOAT is an **IDL_TYP_FLOAT** scalar keyword. It does not use the **specified** field of the **IDL_KW_PAR** struct to get notification of whether the keyword is present. Instead, it uses the **IDL_KW_ZERO** flag to make sure that the variable **f** is always zeroed. If the keyword is present, the value will be written into **f**, otherwise it will remain 0. The important point is that the routine can’t tell the difference between the keyword being absent, or being present with a user-supplied value of zero. If this distinction doesn’t matter, such as when the keyword is to serve as an on/off toggle, use this method. If it does matter, use the specified field as demonstrated with the DOUBLE keyword, above.

23 – 24

LONG is a scalar keyword of **IDL_TYP_LONG**. It sets the **IDL_KW_ZERO** flag to get the variable **l** zeroed prior to keyword parsing. The use of the **IDL_KW_VALUE** flag indicates that if the keyword is present, the value 15 (the lower 12 bits of the flags field) will be ORed into the variable **l**.

25 – 26

The **IDL_KW_OUT** flag indicates that the routine wants gets the **IDL_VPTR** for READWRITE if it is present. Since **IDL_KW_ZERO** is also set, the variable **var** will be zero unless the keyword is present. The specification of **IDL_TYP_UNDEF** here indicates that there is no type conversion or processing applied to **IDL_KW_OUT** keywords.

27

This keyword is included here to force the need for **IDL_KWCleanup()** on line 58.

28

Every array of **IDL_KW_PAR** structs must end with a NULL entry.

31

Mark the stack in preparation for the **IDL_KWCleanup()** call on line 58.

33

Do the keyword processing. The first three arguments are simply the arguments the interpreter passed to the routine. The **plain_args** argument is set to NULL because this routine is registered as not accepting any plain arguments. Since no plain arguments will be present, the return value from IDL_KWGetParams() is discarded.

35

The **l** variable will be 0 if LONG is not present, and 1 if it is.

36

The **f** variable will always have some usable value, but if it is zero there is no way to know if the keyword was actually specified or not.

37 – 38

These keywords use the variables from the specified field of their **IDL_KW_PAR** struct to determine if they were specified or not. Use of the **IDL_STRING_STR** macro is described in "Accessing IDL_STRING Values" on page 329.

39– 45

Accessing the ARRAY keyword is simple. The **arr_there** variable indicates if the keyword is present, and **arr_d.n** gives the number of elements.

47 – 55

Since the READWRITE keyword is accessed via the argument's **IDL_VPTR**, we use the **IDL_Print()** function to print its value. This has the same effect as using the user-level PRINT procedure when running IDL. See "Output of IDL Variables" on page 384. Then, we change its value to 42 using **IDL_StoreScalar()**.

Again, please note that we use this mechanism in order to create a simple example. You will probably want to avoid the use of this type of output (**printf** and **IDL_PRINT()**) in your own code.

57

The use of **IDL_KWCleanup()** is necessitated by the existence of the STRING keyword, which is of **IDL_TYP_STRING**.

Interfaces Obsoleted in IDL 5.6

Changes were required to implement the ability to enable and disable IDL system routines from runtime and callable IDL. Rather than alter the IDL_SYSFUN_DEF structure, and the IDL_AddSystemRoutine() function in an incompatible way, a new structure (IDL_SYSFUN_DEF2) and new function (IDL_SysRtnAdd()) have been created to accomplish the new tasks, and the old structure and function have been obsoleted.

Note

The interfaces described in this section are considered functionally obsolete although they continue to be supported by RSI. This section is supplied to help those maintaining older code. New code should be written using the information found in "Registering Routines" on page 438.

Registering Routines

The **IDL_AddSystemRoutine()** function adds system routines to IDL's internal tables of system functions and procedures. As a programmer, you will need to call this function directly if you are linking a version of IDL to which you are adding routines, although this is very rare and not considered to be a good practice for maintainability reasons. More commonly, you use **IDL_AddSystemRoutine()** in the **IDL_Load()** function of a Dynamically Loadable Module (DLM).

Note

LINKIMAGE or DLMs are the preferred way to add system routines to IDL because they do not require building a separate IDL program. These mechanisms are discussed in the following sections of this chapter.

```
int IDL_AddSystemRoutine(IDL_SYSFUN_DEF *defs, int is_function,
int cnt);
```

It returns *True* if it succeeds in adding the routine or *False* in the event of an error:

defs

An array of **IDL_SYSFUN_DEF** structures, one per routine to be declared. This array must be defined with the C language static storage class because IDL keeps pointers to it. **defs** must be sorted by routine name in ascending lexical order.

is_function

Set this parameter to IDL_TRUE if the routines in **defs** are functions, and IDL_FALSE if they are procedures.

cnt

The number of **IDL_SYSFUN_DEF** structures contained in the **defs** array.

The definition of **IDL_SYSFUN_DEF** is:

```
typedef IDL_VARIABLE *(* IDL_FUN_RET)();

typedef struct {
  IDL_FUN_RET funct_addr;
  char *name;
  UCHAR arg_min;
  UCHAR arg_max;
  UCHAR flags
} IDL_SYSFUN_DEF;
```

IDL_VARIABLE structures are described in "The IDL_VARIABLE Structure" on page 267.

funct_addr

Address of the function implementing the system routine.

name

The name by which the routine is to be invoked from within IDL. This should be a pointer to a null terminated string. The name should be capitalized. If the routine is an object method, the name should be fully qualified, which means that it should include the class name at the beginning followed by two consecutive colons, followed by the method name (e.g. `CLASS::METHOD`).

arg_min

The minimum number of arguments allowed for the routine.

arg_max

The maximum number of arguments allowed for the routine. If the routine does not place an upper value on the number of arguments, use the value **IDL_MAXPARAMS**.

flags

A bitmask that provides additional information about the routine. Its value can be any combination of the following values (bitwise OR'd together to specify more than one at a time) or zero if no options are necessary:

IDL_SYSFUN_DEF_F_OBSOLETE

IDL should issue a warning message if this routine is called and !WARN.OBS_ROUTINE is set.

IDL_SYSFUN_DEF_F_KEYWORDS

This routine accepts keywords as well as plain arguments.

Simplified Routine Invocation

Note

The functions and techniques described in this section are no longer widely used, and are considered functionally obsolete although they continue to be supported by RSI. This section is supplied to help those maintaining older code. New code should be written using the information found in Chapter 21, “Adding System Routines”.

A great deal of the work involved in writing IDL system routines involves checking positional arguments, screening out illegal combinations of type and structure, and converting them to desired type. The function **IDL_EzCall()** provides a simplified way to handle this task. It processes an array of **IDL_EZ_ARG** structs which describe the processing to be applied to each positional argument.

The **IDL_EzCall()** function is similar to the facility provided for keyword arguments by **IDL_KWGetParams()**:

```
void IDL_EzCall(int argc, IDL_VPTR argv[],
                IDL_EZ_ARG arg_struct[]);
```

where:

argc

The number of positional arguments present.

argv

An array of pointers to the positional arguments.

arg_struct

An array of **IDL_EZ_ARG** structures defining the desired characteristics for each possible argument. Note that this array must have a definition for every possible parameter whether that argument is present in the current call or not. The order of the **IDL_EZ_ARG** structures is the same as the order in which the arguments are specified in the call. (See “The IDL_EZ_ARG struct” on page 535.)

There are some things you need to be aware of when using **IDL_EzCall()**:

- **IDL_EzCall()** automatically excludes file variables (such as those created by the ASSOC function) so you don’t have to take any special action to screen such variables out.

- Every call to **IDL_EzCall()** must have a matching call to **IDL_EzCallCleanup()** before execution returns to the interpreter.
- **IDL_EzCall()** does not handle keyword arguments. If the calling routine allows keyword arguments, it must do its own keyword processing using **IDL_KWGetParams()** (see "IDL Internals: Keyword Processing" on page 297) and pass an **argv** containing only positional arguments to **IDL_EzCall()**.
- If you mark a variable as being write-only, you shouldn't count on anything useful being in the **uargv** or **value** fields. This implies that it is not a good idea to set the **IDL_EZ_POST_WRITEBACK** field in the post field. Instead, you will have to allocate a new temporary variable, place the desired value into it, and use the **IDL_VarCopy()** function to write its value back into the original **argv** entry yourself.

Note

IDL_EZ_POST_WRITEBACK is only useful when the access field is set to **IDL_EZ_ACCESS_RW**.

The IDL_EZ_ARG struct

The **IDL_EZ_ARG** struct has the following definition:

```
typedef struct {
  short allowed_dims;
  short allowed_types;
  short access;
  short convert;
  short pre;
  short post;
  IDL_VPTR to_delete;
  IDL_VPTR uargv;
  IDL_ALLTYPES value;
} IDL_EZ_ARG;
```

where:

allowed_dims

A bit mask that specifies the allowed dimensions. Bit 0 means scalar, bit 1 means one-dimensional, etc. The **IDL_EZ_DIM_MASK** macro can be used to specify certain bits. It accepts a single argument that specifies the number of dimensions that are accepted, and returns the bit value that represents that number. For example, to specify that the argument can be scalar or have 2 dimensions:

```
    IDL_EZ_DIM_MASK(0) | IDL_EZ_DIM_MASK(2)
```

In addition, the following constants are defined to simplify the writing of common cases:

IDL_EZ_DIM_ARRAY

Allow all but scalar.

IDL_EZ_DIM_ANY

Allow anything.

allowed_types

This is a bit mask defining the allowed data types for the argument. To convert type codes to the appropriate bits, use the formula:

$$\text{BitMask} = 2^{\text{TypeCode}}$$

or use the **IDL_TYP_MASK** macro (see "Type Masks" on page 259).

Note

If you specify a value for the convert field, its a good idea to specify **IDL_TYP_B_ALL** or **IDL_TYP_B_SIMPLE** here. The type conversion will catch any problems and your routine will be more flexible.

access

A bitmask that describes the type of access to be allowed to the argument. The following constants should be OR'd together to set the proper value:

IDL_EZ_ACCESS_R

The value of the argument is used by the system routine.

IDL_EZ_ACCESS_W

The value of the argument is changed by the system routine. This means that it must be a named variable (as opposed to a constant or expression).

IDL_EZ_ACCESS_RW

This is equivalent to **IDL_EZ_ACCESS_R** | **IDL_EZ_ACCESS_W**.

convert

The type code for the type to which the argument will be converted. A value of **IDL_TYP_UNDEF** means that no conversion will be applied.

pre

A bitmask that specifies special purpose processing that should be performed on the variable by **IDL_EzCall()**. These bits are specified with the following constants:

IDL_EZ_PRE_SQMATRIX

The argument must be a square matrix.

IDL_EZ_PRE_TRANSPOSE

Transpose the argument.

Note

This processing occurs after any type conversions specified by **convert**, and is only done if the access field has the **IDL_EZ_ACCESS_R** bit set.

post

A bit mask that specifies special purpose processing that should be performed on the variable by **IDL_EzCallCleanup()**. These bits are specified with the following constants:

IDL_EZ_POST_WRITEBACK

Transfer the contents of the **uargv** field back to the actual argument.

IDL_EZ_POST_TRANSPOSE

Transpose **uargv** prior to transferring its contents back to the actual argument.

Note

This processing occurs only when the **access** field has the **IDL_EZ_ACCESS_W** bit set. If **IDL_EZ_POST_WRITEBACK** is not present, none of the other actions are considered, since that would imply wasted effort.

to_delete

Do not make use of this field. This field is reserved for use by the EZ module. If **IDL_EzCall()** allocated a temporary variable to satisfy the conversion requirements given by the convert field, the **IDL_VPTR** to that temporary is saved here for use by **IDL_EzCallCleanup()**.

uargv

After calling **IDL_EzCall()**, **uargv** contains a pointer to the **IDL_VARIABLE** which is the argument. This is the **IDL_VPTR** that your routine should use. Depending on the required type conversions, it might be the actual argument, or a temporary variable containing a converted version of the original. This field won't contain anything useful if the **IDL_EZ_ACCESS_R** bit is not set in the **access** field.

value

This is a copy of the **value** field of the **IDL_VARIABLE** pointed at by **uargv**. For scalar variables, it contains the value, for arrays it points at the array block. This field is here to make reading read-only variables faster. Note that this is only a copy from **uargv**, and changing it will not cause the actual **value** field in **uargv** to be updated.

Cleaning Up

Every call to **IDL_EzCall()** must be bracketed by a call to **IDL_EzCallCleanup()**:

```
void IDL_EzCallCleanup(int argc, IDL_VPTR argv[],
                       IDL_EZ_ARG arg_struct[]);
```

The arguments are exactly the same as those passed to **IDL_EzCall()**.

Example— using IDL_EzCall()

The following function skeleton shows how to use the simplified interface to handle argument processing for an older version of the built-in SVD (Singular Value Decomposition) function. SVD accepts the following positional arguments (in order):

A

An *m* by *n* matrix (input, required).

W

An *n*-element vector (output, required).

U

An *n* by *m* matrix (output, optional)

V

An *n* by *n* matrix (output, optional)

Each line is numbered to make discussion easier. These numbers are not part of the actual program.

```
1  void nr_svdcmp(int argc, IDL_VPTR argv[])
2  {
3    .
4    .
5    .
6    static IDL_EZ_ARG arg_struct[] = {
7      { IDL_EZ_DIM_MASK(2), IDL_TYP_B_SIMPLE, IDL_EZ_ACCESS_R,
8        IDL_TYP_FLOAT, 0, 0 }, /* A */
9      { IDL_EZ_DIM_ANY, IDL_TYP_B_ALL,
10       IDL_EZ_ACCESS_W, 0, 0, 0 }, /* w */
11     { IDL_EZ_DIM_ANY, IDL_TYP_B_ALL,
12       IDL_EZ_ACCESS_W, 0, 0, 0 }, /* U */
13     { IDL_EZ_DIM_ANY, IDL_TYP_B_ALL,
14       IDL_EZ_ACCESS_W, 0, 0, 0 } /* V */
15   };
16
17   IDL_EzCall(argc, argv, arg_struct);
18   .
19   .
20   .
21   /* Do the SVD calculation and prepare temporary
22      variables to be returned as w, U, and V */
23   .
24   .
25   .
26   IDL_EzCallCleanup(argc, argv, arg_struct);
27 }
```

Those features of this procedure that are interesting in terms of plain argument processing are, by line number:

7-8

The settings of the various fields of the **IDL_EZ_ARG** struct for the first positional argument (A) specifies:

allowed_dims

The argument must be 2-dimensional.

allowed_types

It can have any simple type. Types and type codes are discussed in "IDL Internals: Types" on page 257.

access

The routine will examine the argument's value, but will not attempt to change it.

convert

The argument should be converted to **IDL_TYP_FLOAT** if necessary.

pre

No pre-processing is required.

post

No post-processing is required.

...

The remaining fields are all set by **IDL_EzCall()** in response to the above.

9-14

Arguments two through four are allowed to have any number of dimensions and are allowed any type. This is because the routine does not intend to examine them, only to change them. For the same reason, a zero (**IDL_TYP_UNDEF**) is specified for the convert field indicating that no type conversion is desired. No pre or post-processing is specified.

17

Process the positional arguments.

26

Clean up.

Obsolete Error Handling API

The following variables can be accessed only on UNIX. These variables have been superseded by the functions listed in "Functions for Returning System Variables" on page 401, which are available on all platforms. In all cases, these variables should be considered READ-ONLY:.

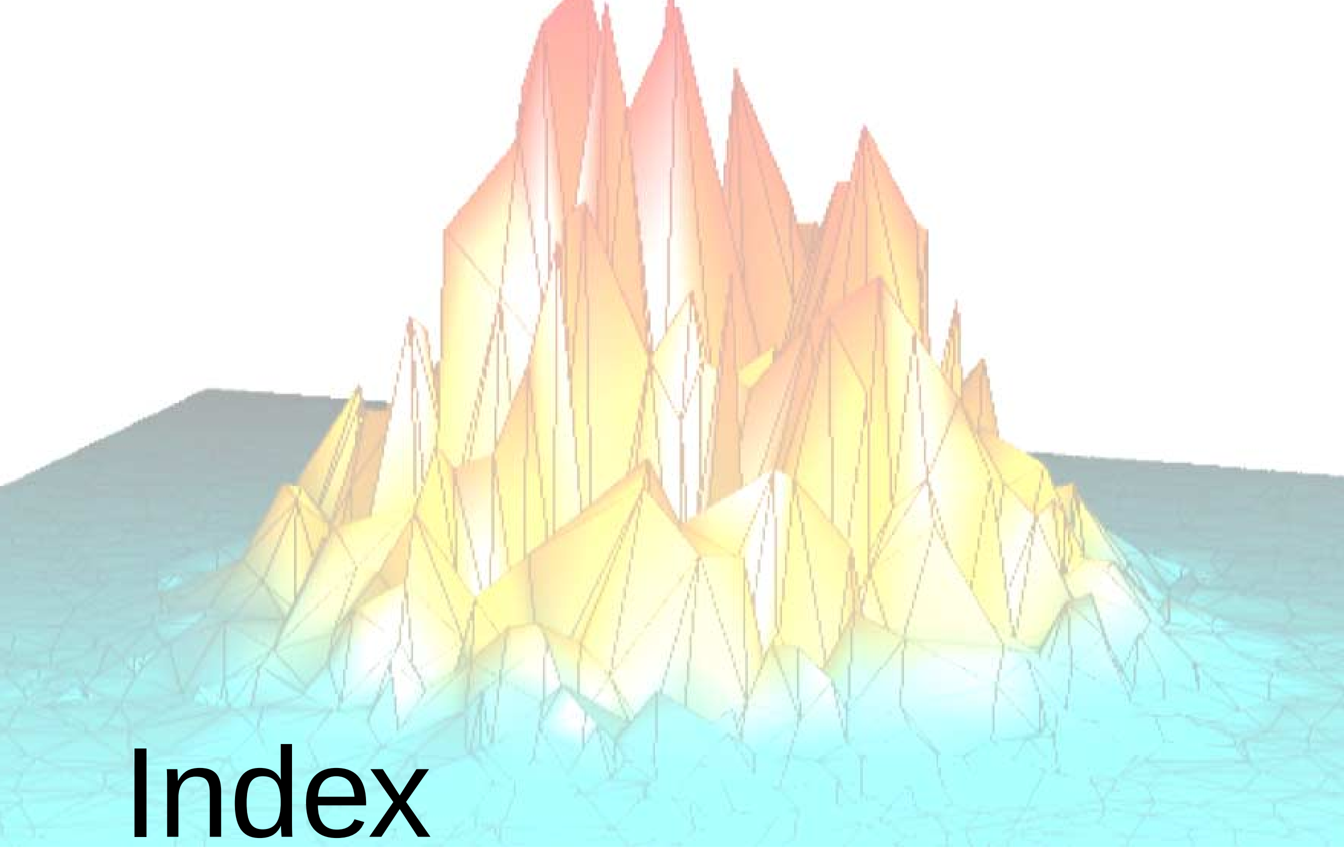
IDL System Variable	Internal Variable	Type
!DIR	IDL_SysvDir	IDL_STRING
!VERSION.ARCH	IDL_SysvVersion.arch	IDL_STRING
!VERSION.OS	IDL_SysvVersion.os	IDL_STRING
!VERSION.OS_FAMILY	IDL_SysvVersion.os_family	IDL_STRING
!VERSION.RELEASE	IDL_SysvVersion.release	IDL_STRING
!ERR	IDL_SysvErrCode	IDL_LONG
!ERROR	IDL_SysvErrorCode	IDL_LONG
!ORDER	IDL_SysvOrder	IDL_LONG

Table A-1: IDL System Variables Available to User Programs

In addition, the following function has been superseded by the IDL_SysvErrorCodeValue( ) function:

IDL_LONG IDL_SysvErrCodeValue(void)

This function returns the value of !ERR.



Index