IDL®

External Development Guide

RESEARCH SYSTEMS
Software ≡ Vision™

IDL Version 5.3
September, 1999 Edition

Acknowledgments

IDL® is a trademark 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.

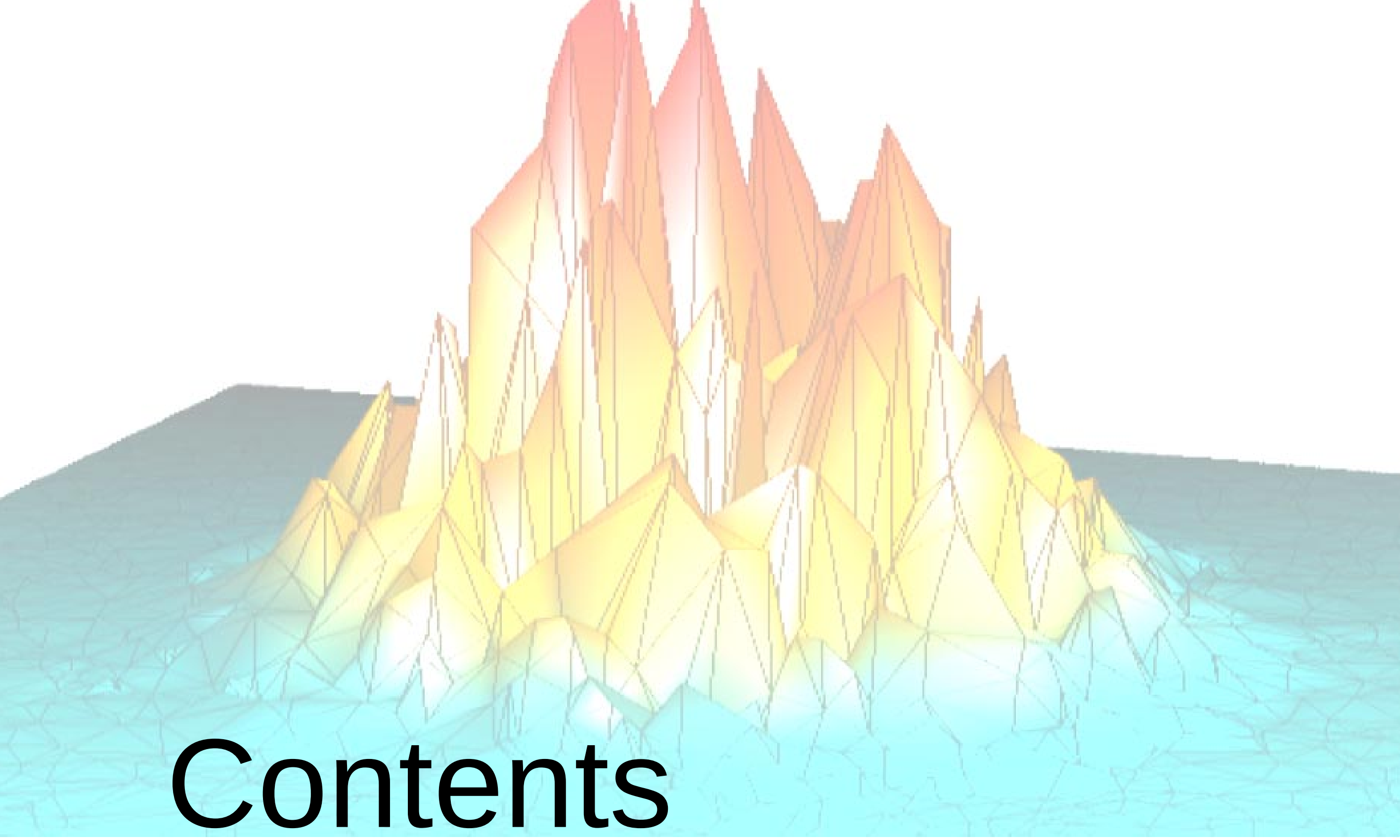
GRG2™ is a trademark of Windward Technologies, Inc. The GRG2 software for nonlinear optimization is used by permission.

NCSA Hierarchical Data Format (HDF) Software Library and Utilities
Copyright 1988-1998 The Board of Trustees of the University of Illinois
All rights reserved.

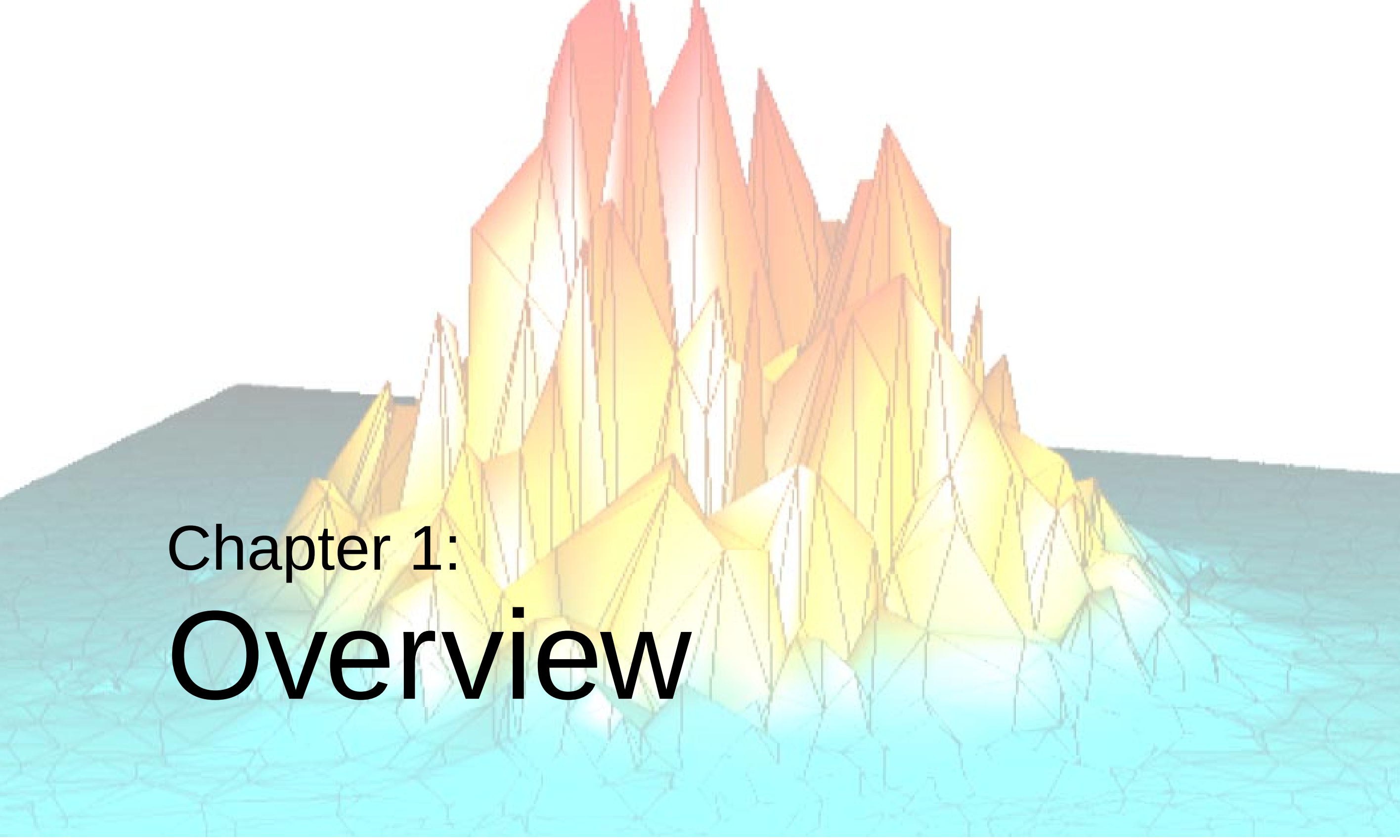
Portions of this software are copyrighted by INTERSOLV, Inc., 1991-1998.

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

Research Systems, Inc. documentation is printed on recycled paper. Our paper has a minimum 20% post-consumer waste content and meets all EPA guidelines.



Contents



Chapter 1: Overview

This chapter discusses the following topics:

About this Manual

This manual describes the internal implementation of IDL in sufficient detail to allow the user to write code in other languages and link it with IDL. It explains how to write code that will be called directly from IDL (CALL_EXTERNAL), how to add built-in system routines and functions (LINKIMAGE), and how to call IDL as a subroutine from other programs ("Callable IDL").

Using this Document with Previous Versions of IDL

This document describes the interface to the IDL internals introduced with IDL version 4.0. Those using CALL_EXTERNAL or LINKIMAGE with previous versions of IDL should consult Appendix A , "Obsolete Internal Interfaces" for compatibility information.

Inter-language Communication Techniques Which are Supported

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. Under UNIX and VMS, 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, “SPAWN”, or in the documentation for SPAWN in the *IDL Reference Guide*.

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, and data can only be returned by writing it to a temporary file which IDL subsequently opens and reads.
- Macintosh or Windows systems are unable to capture returned data via the Result parameter, further reducing flexibility.

Recommendation

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

ActiveX

IDLDrawWidget is an OLE custom control (OCX) built around IDL for Windows that provides an easy mechanism for integrating IDL with Microsoft Windows 95 and NT applications written in languages such as C, C++, Visual Basic, Fortran, Delphi, and others. This is the most natural way to combine IDL with such environments, and is therefore the easiest option under Windows. For more information, see Chapter 3, “IDLDrawWidget ActiveX Control”.

Advantages

- Integrates easily with an important interprocess communication mechanism under Windows 95/98 and NT.
- Higher level than the function call interfaces supported by the remaining options.

- Uses native syntax in languages such as Visual Basic, Visual C++, and Delphi.

Disadvantages

- Only supported under Microsoft Windows.

Recommendation

Use the IDL ActiveX control if you are writing a Windows-only application written 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.

AppleScript

On the Macintosh, IDL can act as an AppleScript server or client.

Advantages

- Integrates easily with standard Apple interprocess communication mechanism on the Macintosh.
- Higher level than the function call interfaces supported by the remaining options.
- Far more capable than SPAWN on the Macintosh, allowing remote control of any scriptable application, the system, and finder.
- Allows import/export of data from IDL.

Disadvantages

- Only supported on the Macintosh.
- Not possible to integrate graphics from IDL into another program's drawing area.

Recommendation

AppleScript is excellent for scripting operations, more sophisticated than what SPAWN allows, but less capable than the IDL ActiveX control available under Microsoft Windows.

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 6, "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 LINKIMAGE 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 7, "CALL_EXTERNAL" and the documentation for CALL_EXTERNAL in the *IDL Reference Guide*.

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 (i.e. without knowledge of IDL internals). For safety, you should write 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 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, Dynamically Loadable Modules)

It is possible to merge routines written in other languages with IDL at run-time. Such routines are dynamically linked, as with CALL_EXTERNAL. They are more difficult to write, but more flexible and powerful. LINKIMAGE provides access to variables and other objects inside of IDL.

This book contains the information necessary to successfully add your own code to IDL using LINKIMAGE. Especially important is Chapter 18, “Adding System Routines”. Additional information about LINKIMAGE can be found in Chapter 7, “CALL_EXTERNAL” and in the documentation for LINKIMAGE in the *IDL Reference Guide*.

Advantages

- This is the most fully integrated option. It allows writing 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.

Recommendation

Use LINKIMAGE 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 for Windows, IDL for UNIX, and IDL for VMS are 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 called “Callable IDL” to distinguish it from the more usual case of calling your code *from* IDL via CALL_EXTERNAL or LINKIMAGE. IDL for Macintosh supports “calls” to IDL via AppleScript.

This book contains the information necessary to successfully call IDL from your own code. Introductory material and caveats about Callable IDL can be found in Chapter 1, “Overview”.

Advantages

- Supported on almost 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.
- Not available on the Macintosh.

Recommendation

Most platforms offer a specialized method to call other programs that might be more appropriate. Windows 95 or NT users should consider the ActiveX control. UNIX users should consider using the IDL RPC server. Macintosh users do not have Callable IDL available and should use AppleScript. 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 Dynamic Linking 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:

- Macintosh: Code Fragments
- UNIX: Sharable Libraries
- VMS: Sharable Libraries and Sharable Executables
- 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 built in 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:

- Research Systems 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 Research Systems, 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 Research Systems 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:

ActiveX

To use the IDL ActiveX control, you should be familiar with the programming environment in which you will be using the control (e.g. Visual Basic). A lower level understanding of ActiveX and COM is not necessary, but might be useful.

RPC

To use IDL as an RPC server, a knowledge of Sun RPC (Also known as ONC RPC) is required. The Sun 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

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

Research Systems to actively support 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 ends 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. Here are some issues to consider. 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 (Active X, AppleScript) exist on some systems that simplify the details of inter-language linking.

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, WIN32S, 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.

OpenVMS

You should understand system services, the Run Time Libraries, file I/O, and processes.

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 95.* Redmond, Washington: Microsoft Press, 1996. ISBN 1-55615-676-6

Richter, Jeffery. *Advanced Windows, Third Edition.* Redmond, Washington: Microsoft Press, 1997. 1-57231-548-2

Microsoft, 1993. *Win32 Programmers Reference* Volumes 1-5. Redmond, Washington: Microsoft Press, 1993. ISBN 1-55615-515-8 (v.1), ISBN 1-55615-516-6 (v.2), ISBN 1-55615-517-4 (v.3), ISBN 1-55615-518-2 (v.4), ISBN 1-55615-519-0 (v.5),

Microsoft, 1997. *Microsoft Visual C++ Reference* Volumes 1-4. Redmond Washington: Microsoft Press, 1997. ISBN 1-57231-518-0 (v.1), ISBN 1-57231-519-9 (v.2), ISBN 1-57231-520 (v.3), ISBN 1-57231-521-0 (v.4).

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 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.

OpenVMS

The standard OpenVMS reference manuals published by Digital Equipment Corporation cover the material needed by OpenVMS programmers; of special importance are the System Services, Run Time Libraries, C Language, and Linker documentation.

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.)

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 for the reasons stated below.

- 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 169). 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 `export.h`, found in the `external` 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

`export.h` contains some declarations which are necessary to the compilation process, but which are still considered private to Research Systems. 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 `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 `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 "export.h"
```

Linking Details

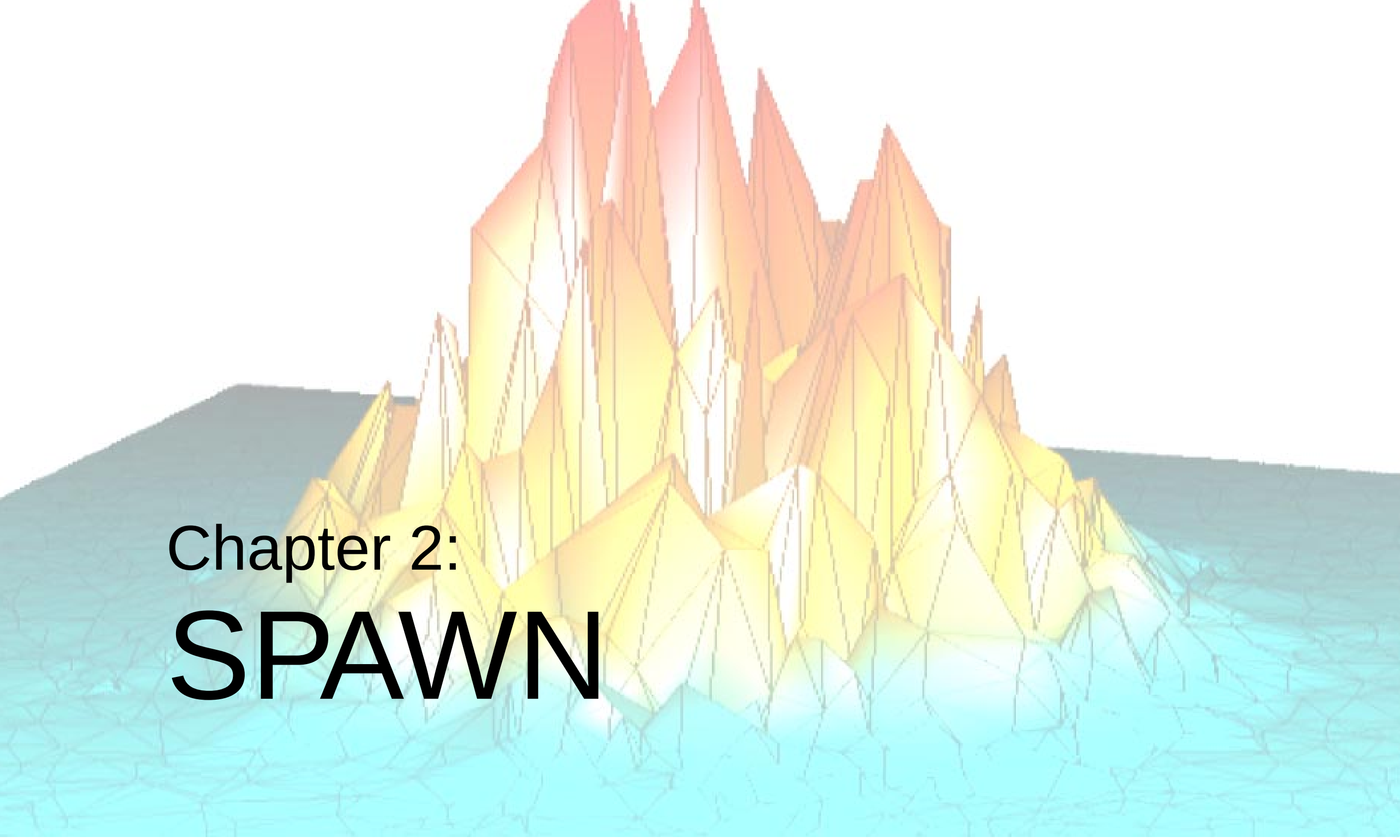
Once you’ve written your code, you need to compile it and link it into IDL before it can be run. Information on how to do this is available in the various subdirectories of the external subdirectory of the IDL distribution. References to files that are useful in specific situations are contained in this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

In addition:

- The 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 360.

Reading the Remainder of this Book

After reading this chapter, the remainder of this book can be read in many different orders. Chapters 2 through 11 can be read in any order and are largely independent of each other. Those wishing to add system routines to IDL will need to read Chapter 18, “Adding System Routines”, while those wishing to use Callable IDL should start with Chapter 19, “Introduction to Callable IDL”.



Chapter 2: SPAWN

This chapter discusses the following topics:

The SPAWN Procedure

The IDL SPAWN procedure spawns a child process to execute a command or series of commands. Under UNIX, the shell used (if any) is obtained from the SHELL environment variable. Under OpenVMS, the DCL command language interpreter is used. Under Windows, a DOS window or Command Shell is opened. SPAWN can also create an interactive command interpreter process. On the Macintosh, SPAWN opens specified files or applications.

The SPAWN procedure has the following syntax:

SPAWN[, *Command*[, *Result*]]

where

Command

A string containing the command or commands to be issued. If *Command* is not present, SPAWN starts an interactive command interpreter process if possible.

Command must be of type string. Under OpenVMS, it is restricted to being a scalar. Under UNIX, it can be a string array (each element is passed to the child process as a separate argument) if used in conjunction with the NOSHELL keyword. If a new UNIX shell process is started (that is, if the NOSHELL keyword is *not* specified), *Command* must be a scalar string.

On the Macintosh, *Command* must consist of the names of files to be opened. Multiple filenames can be entered. If the first filename is an application, it is used to open the remaining files. Otherwise, each file is opened by the application that owns it. IDL execution resumes when the files have been opened. Interactive access to the Macintosh command interpreter is not supported.

Result

If *Result* is not present, the output from the child process simply goes to the standard output (usually the terminal). Otherwise, the output from the child process is placed into a string array (one line of output per array element) and assigned to *Result*.

Under Windows and the Macintosh OS, specifying *Result* has no effect.

Interactive Use of SPAWN

If SPAWN is called without arguments, an interactive command interpreter process is started. The user can enter one or more operating system commands. While you use the command interpreter process, IDL is suspended. When you exit the child process, control returns to IDL, which resumes at the point where it left off. The IDL session remains exactly as you left it.

It should be noted that using SPAWN in this manner is equivalent to using the IDL "$" command. The difference between these two is that "$" can only be used interactively while SPAWN can be used interactively or in IDL programs.

UNIX Command Interpreter

UNIX offers many shells. The two most common are the Bourne shell (`/bin/sh`) and the C shell (`/bin/csh`). Rather than force use of a given shell, IDL follows the UNIX convention of using the shell specified by the UNIX environment variable *SHELL*. If *SHELL* does not exist, the Bourne shell is used.

Under UNIX, the interactive form of SPAWN is provided primarily for users of the Bourne shell and for compatibility with OpenVMS. Shells that offer process suspension (e.g., `/bin/csh`) offer a more convenient and efficient way to get the same effect.

The following statements demonstrate the use of SPAWN on a UNIX system:

```
SPAWN
% date
Fri Aug 26 13:55:00 MDT 1998
% exit
```

OpenVMS Command Interpreter

Under OpenVMS, the command interpreter used is always DCL. If you specify the NOWAIT keyword to SPAWN, the IDL process is not suspended until the spawned process completes. The following statements demonstrate SPAWN under OpenVMS:

```
SPAWN
$ SHOW TIME
29-JAN-1998 16:32:23
$ LOGOUT
```

Windows Command Interpreter

Under Windows 95/98, the command interpreter used is always `COMMAND.COM`; under Windows NT, `CMD.EXE` is used. Calling SPAWN (or $) with no arguments creates an interactive command interpreter window as a child process.

Macintosh Command Interpreter

Interactive access to the Macintosh command interpreter is not supported.

Noninteractive Use of SPAWN

If SPAWN is called with a single argument, that argument is taken as a command to be executed. In this case, IDL starts a child command interpreter process and passes the command to it. The argument should be a scalar string. The shell executes the command and exits, at which point IDL resumes operation. This form of operation is very convenient for executing single commands from IDL programs. For example, it is sometimes useful to create a temporary scratch file. SPAWN can be used as demonstrated in the following program fragment. First, open a scratch file. Use the GET_LUN keyword to allocate a file unit.

```
OPENW, UNIT, 'scratch.dat', /GET_LUN

;...IDL commands go here.

;Deallocate the file unit and close the file.
FREE_LUN, UNIT

;Use the !VERSION system variable to determine the proper file
;deletion command for the current operating system. Since this
;operation is not supported on the Macintosh, jump to an error
;message.
CASE !VERSON.OS OF
  'vms': CMD = 'DELETE'
  'windows': CMD = 'DEL'
  'MacOS': GOTO, NOTSUP
  ELSE: CMD = 'rm'
ENDCASE

;Delete the file using SPAWN.
SPAWN, CMD + 'scratch.dat'

;Jump to the end of the procedure.
GOTO, DONE
NOTSUP: PRINT, 'This operation is not supported on the Macintosh'
DONE:
END
```

Note that the DELETE keyword to the OPEN procedures is a more efficient way to handle this job. The above example should serve only to demonstrate use of the SPAWN procedure.

Macintosh Command Interpreter

You can specify one or more file names when invoking SPAWN on the Macintosh. Each file specified is opened by the application that created it, unless the first file name is that of an application. In this case, the application is used to open the remaining files.

Capturing Output

Under Windows and on the Macintosh, it is not possible to capture output from a spawned command to an IDL variable.

Under UNIX and OpenVMS, by default, any output generated by a spawned command is sent to the standard output, which is usually the terminal. It is possible to capture this output in an IDL string array by calling SPAWN with a second argument. If this second argument, called *Result,* is present, all output from the child process is put into a string array, one line of output per array element, and is assigned to *Result*. For example, the following IDL statements can be used to give a simplistic count of the number of users logged onto a computer running either UNIX or OpenVMS:

```
;Use the !VERSION system variable to determine the command to use.
IF (!VERSION.OS EQ 'VMS') THEN CMD='SHOW USERS' ELSE CMD='who'

;Issue the command, catch the result in a string array.
SPAWN, cmd, users

;Count how many lines of output came back. Under UNIX, this is the
;number of users logged in.
N = N_ELEMENTS(users)

;OpenVMS outputs five extra header lines that are not actual users.
IF (!VERSION.OS EQ 'VMS') THEN N = N - 5

;Print the result.
PRINT, 'There are ', N,' users logged on.'
```

See “SPAWN” in the IDL Reference Guide for further information.

Avoiding the Shell Under UNIX

As mentioned above, SPAWN usually creates a shell process and passes the command to this shell, instead of simply creating a child process to directly execute the command. This default action is taken because the shell provides useful actions such as wildcard expansion and argument processing. Although this is usually desirable, it has the drawback of being slower than necessary. It simply takes longer to start a shell. However, it is possible to avoid using the shell by using the NOSHELL keyword.

When SPAWN is called with the NOSHELL keyword set, the command is executed as a direct child process, avoiding the extra overhead of starting a shell. This is faster; but since there is no shell to break the command into separate arguments, the user has to do it. Every UNIX program is called with a series of arguments. When you issue a shell command, you separate the arguments with white space (blanks and tabs). The shell then breaks up the command into an array of arguments and calls the command (the first word of the command), passing it the array of arguments.

In this case, the *Command* argument should be a string array. The first element of the array is the name of the command to use, and the following elements contain the arguments.

For example, consider the command,

```
SPAWN, 'ps ax'
```

that uses the UNIX *ps* command to show running processes on the computer. To issue this command without a shell, you would write it as follows:

```
SPAWN, ['ps', 'ax'], /NOSHELL
```

Communicating Through the Use of a UNIX Child Process

Using SPAWN in the above examples, the IDL process waited until the child process was finished before continuing. It is also possible to start a child process and immediately continue without waiting for it to finish. In this case, IDL attaches a bidirectional pipe to the standard input and output of the child process. This pipe 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 are 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, the UNIT keyword is used with SPAWN to specify a named variable into which the logical file unit number will be stored. Once the child process has done its work, the FREE_LUN procedure is used 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.
- The EOF function always returns “False” when applied to a pipe. This means that it is not possible to use this function to know when the child process is finished. As a result, the child process must be written in such a way that the controlling IDL procedure knows how much data to send and how much is coming back.
- 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 will have no effect.

Example: Communicating with a Child Process Under UNIX

The C program shown in the following figure (`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. It is simpler and more efficient to perform the calculation within IDL. However, it does serve to illustrate the 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.

C

```
#include <stdio.h>
extern int errno;                      /* System error number */
extern char *sys_errlist[];            /* System error messages */
extern int sys_nerr;                  /* Length of sys_errlist */
main()
{
    float *data, total = 0.0;
    long i, n;
    /* Make sure the output is not buffered */
    setbuf(stdout, (char *) 0);
    /* Find out how many points */
    if (!fread(&n, sizeof(long), 1, stdin)) goto error;
    /* Get memory for the array */
    if (!(data = (float *) malloc((unsigned)
     (n * sizeof(float))))) goto error;
    /* Read the data */
    if (!fread(data, sizeof(float), 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(float), 1, stdout))
     goto error;
     return;
   error:
     fprintf(stderr, "test_pipe: %s\n",
     sys_errlist[errno]);
}
```

Table 2-1: testpipe.c

This program performs the following steps:

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 and use of `EOF` is not necessary.

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: IDLDrawWidget ActiveX Control

This chapter discusses the following topics:

IDLDrawWidget ActiveX Control

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.

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 4, “IDL ActiveX Control Command Reference”.

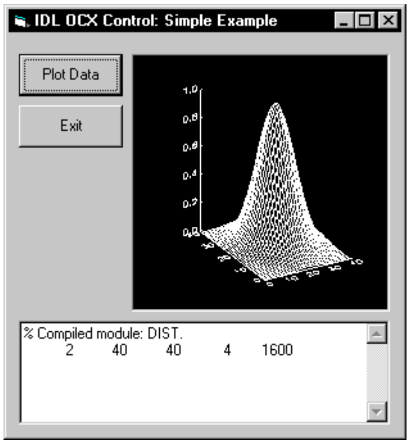
In this chapter, you will be guided through a series of examples designed to demonstrate techniques for integrating IDL with programs written in Microsoft Visual Basic. These techniques begin with writing a simple application that shows how IDL can respond to Visual Basic events and draw graphics in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

Note

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.

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 table:



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

Figure 3-1: 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

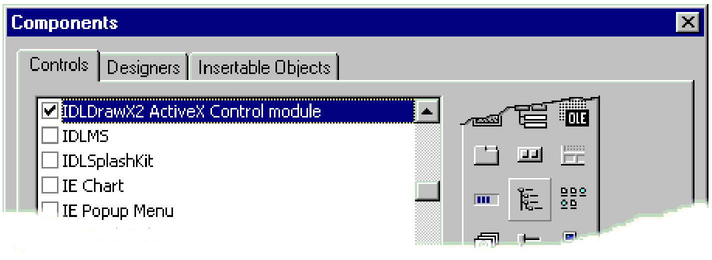
```
Private Sub Form_Load()
    n = IDLDrawWidget1.InitIDL(Form1.hWnd)
    If n <= 0 Then
        MsgBox ("IDL failed to initialize")
        End
    End If
    IDLDrawWidget1.CreateDrawWidget
    IDLDrawWidget1.SetOutputWnd (IDL_Output_Box.hWnd)
End Sub

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 3-1: 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.



[✓] IDLDrawX2 ActiveX Control module
IDLMS
IDLSplashKit
IE Chart
IE Popup Menu

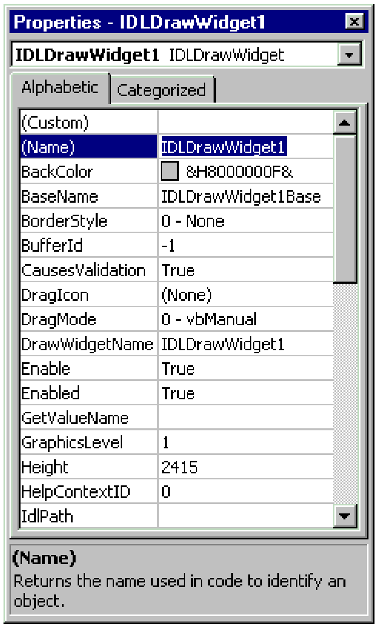
Figure 3-2: List of Available Components

Select the “IDLDrawX2 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

1. 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 4, “IDL ActiveX Control Command Reference” for a complete list of the properties and methods to **IDLDrawWidget**. 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 4, “IDL ActiveX Control Command 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 3-3: 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\idl53
```

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()
```

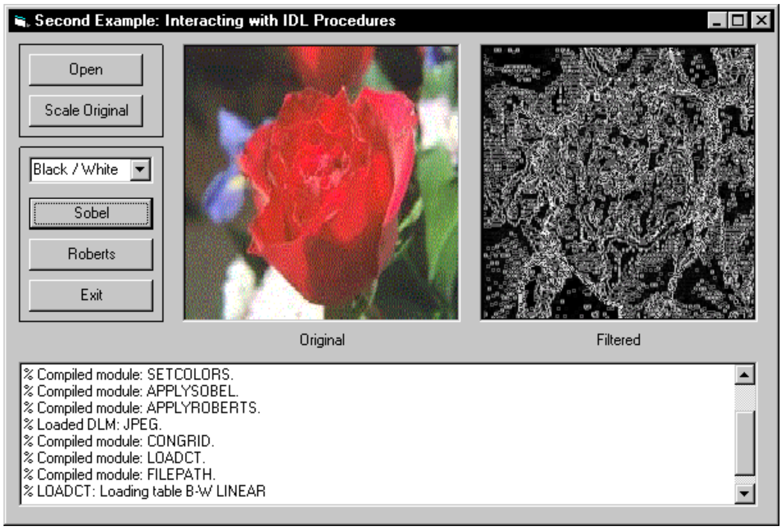
```
        IDLDrawWidget1.DoExit
        End
    End Sub
```

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 following directory:

```
examples\doc\ActiveX\SecondExample.
```



```
% 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 3-4: 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\idl53)` 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
```

Figure 3-5: 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 3-2: 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.

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.

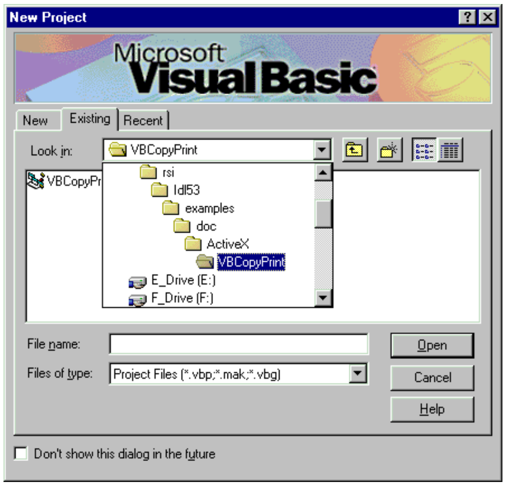


Figure 3-6: Opening the VBCopyPrint project

Running the VBCopyPrint Example

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

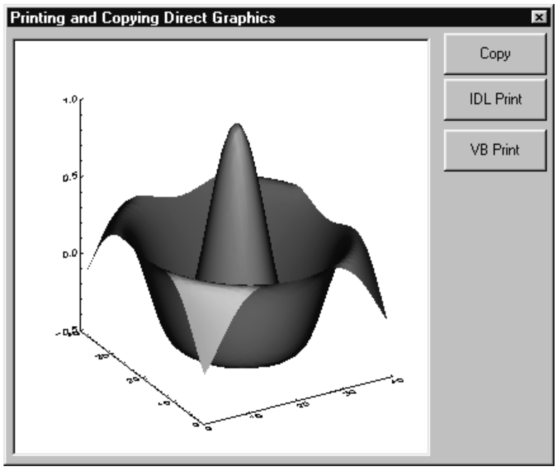


Figure 3-7: VBCopyPrint example

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 3-3: 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** (shown in the following table).

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 3-4: 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,” & Str$(IDLDrawWidget1.DrawId)
5    Screen.MousePointer = vbDefault
6    MsgBox "Window sent to printer."
   End Sub
```

Table 3-5: 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 3-6: VBPrint Command

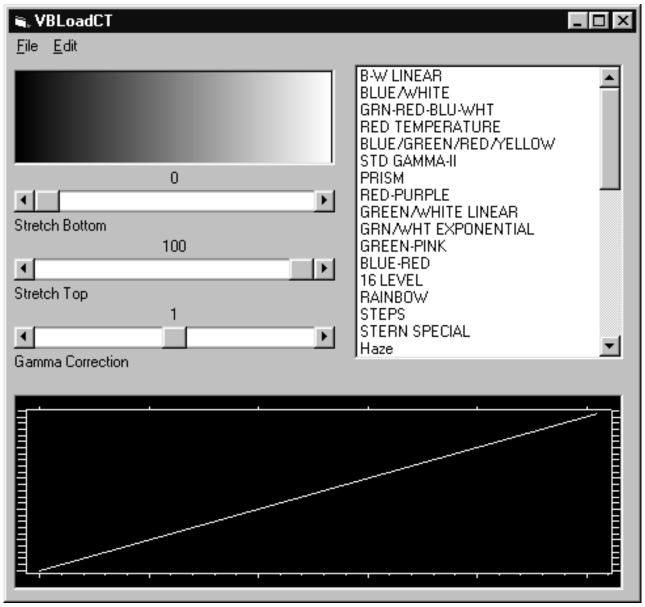
XLoadCT Functionality using Visual Basic

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.



0
Stretch Bottom
100
Stretch Top
1
Gamma Correction

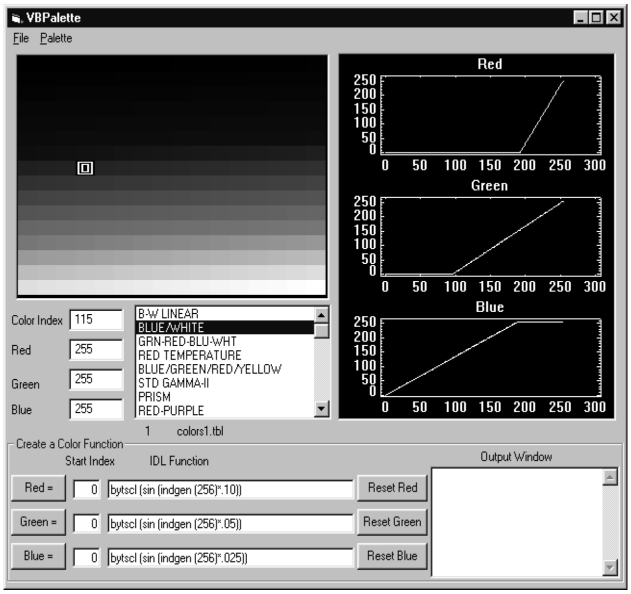
Figure 3-8: VBLoadCT example

XPalette Functionality Using Visual Basic

Like **VBLoadCT**, **VBPalette** demonstrates how to duplicate IDL tool functionality using a Visual Basic interface. See the following figure. 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 3-9: VBPalette Example

Integrating Your Object Graphics by Utilizing 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.

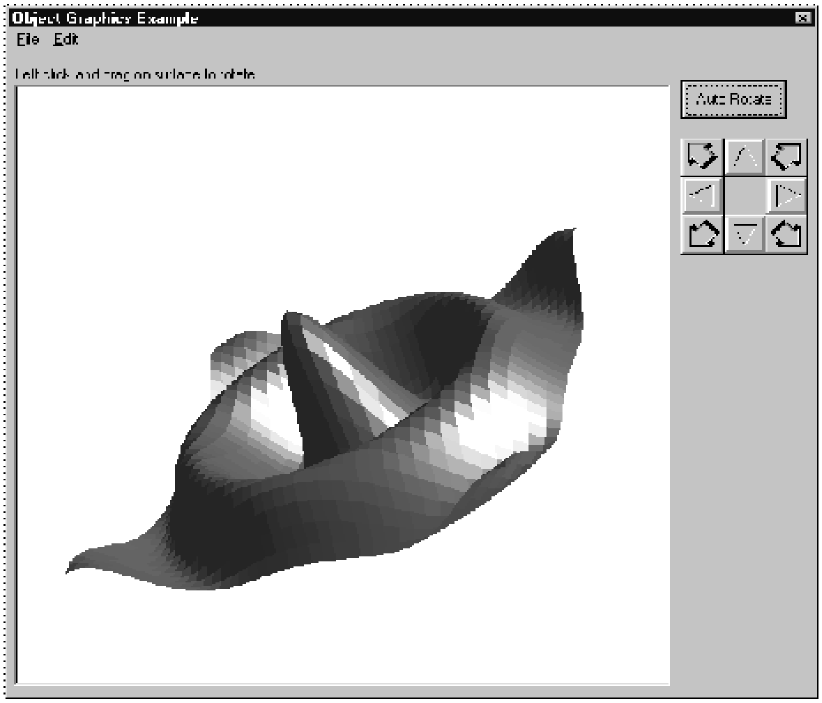


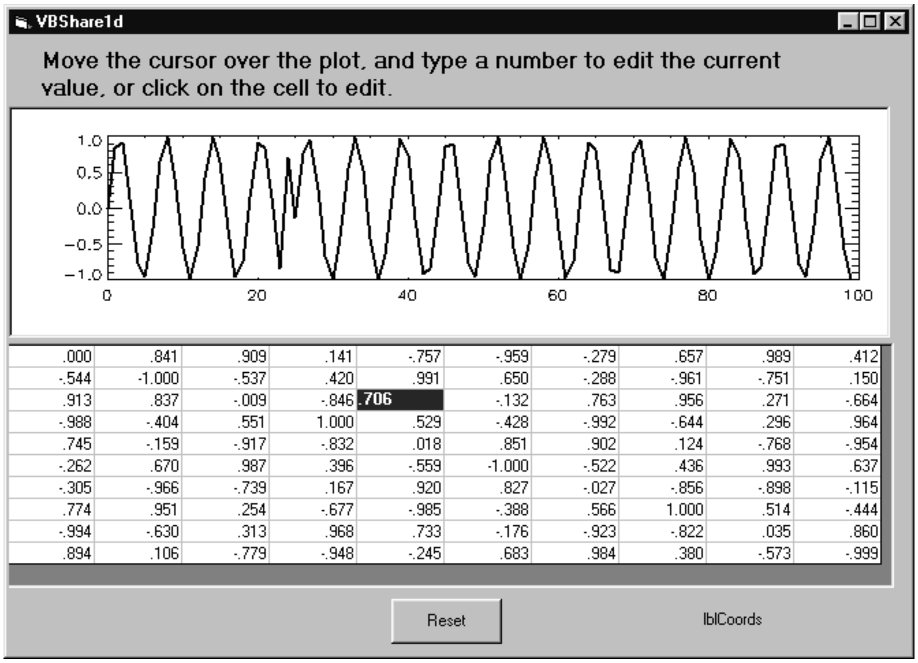
Figure 3-10: VBObjGraph example

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.

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:



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 3-11: VBShare1D

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.

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:

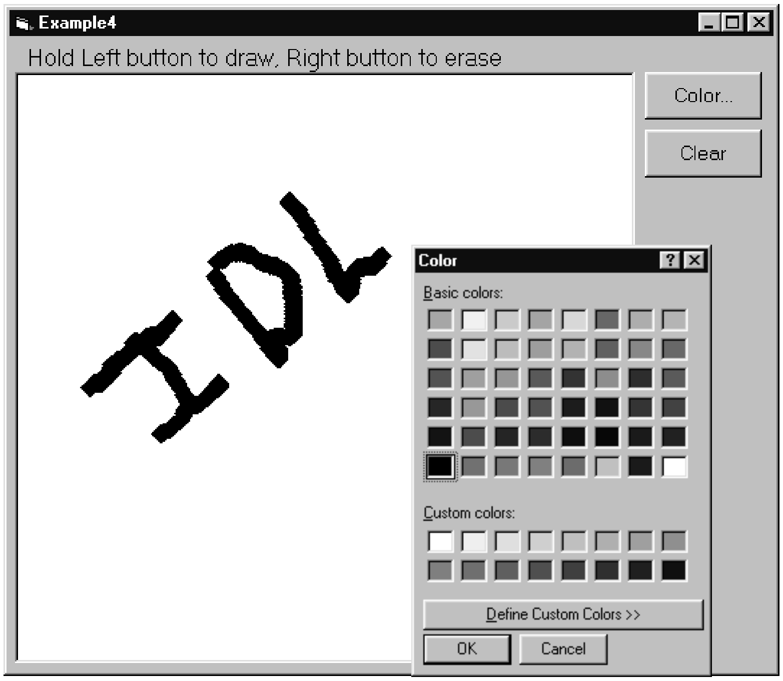


Figure 3-12: VBPaint example

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.

Distributing Your Application

For information on how to distribute an application developed with the IDL ActiveX control, see the *Building IDL Applications* manual.

Chapter 4: IDL ActiveX Control Command 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.

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 4-1: 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

Value	Meaning
-2	IDL is unlicensed (demo)

Table 4-2: 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.

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 the window at screen resolution (72-96 dpi) with a Direct Graphics window. For information about controlling print resolution of an object graphics window, see the BufferId property.

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 4-3: Forwarding events

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.

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 4-4: 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. If the data is an array, the **SetNamedArray** method will be called.

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**.

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 as 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 on IDLgrBuffer, see the *IDL Reference Guide*.

Note

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

The following example shows how to use the new BufferId property:

```
'Create an IDLgrBuffer with dimensions of 1280x1024
IDLDrawWidget1.ExecuteStr("buffer=OBJ_NEW(IDLgrBuffer, $
   dimensions=[1280,1024])")

'Get the object reference of the buffer we just created
buffer=IDLDrawWidget1.GetNamedData("buffer")

'Set the buffer ID to the object reference
IDLDrawWidget1.BufferId=buffer

'Increase the size of the buffer to maximum buffer dimensions
IDLDrawWidget1.ExecuteStr("buffer->SetProperty(dimensions = $
   [1600,1200])")
```

Tip

Remember to destroy the IDLgrBuffer object after it is no longer needed for printing purposes.

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. The **Xsize** property can be set at runtime prior to a call to **CreateDrawWidget**.

Ysize (Design time)

Virtual height of **IDLDrawWidget**. If this value is greater than the **Yviewport** value, scroll bars will be added. The **Ysize** property can be set at runtime prior to a call to **CreateDrawWidget**.

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.

Tip

You can check the return value from the ExecuteStr method to determine if an error occurred. For more information, see IDLgrWindow in the *IDL Reference Guide*.

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.
xPos	The horizontal position of the mouse when the button was clicked.

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

Table 4-5: OnDblClick Parameters

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

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:

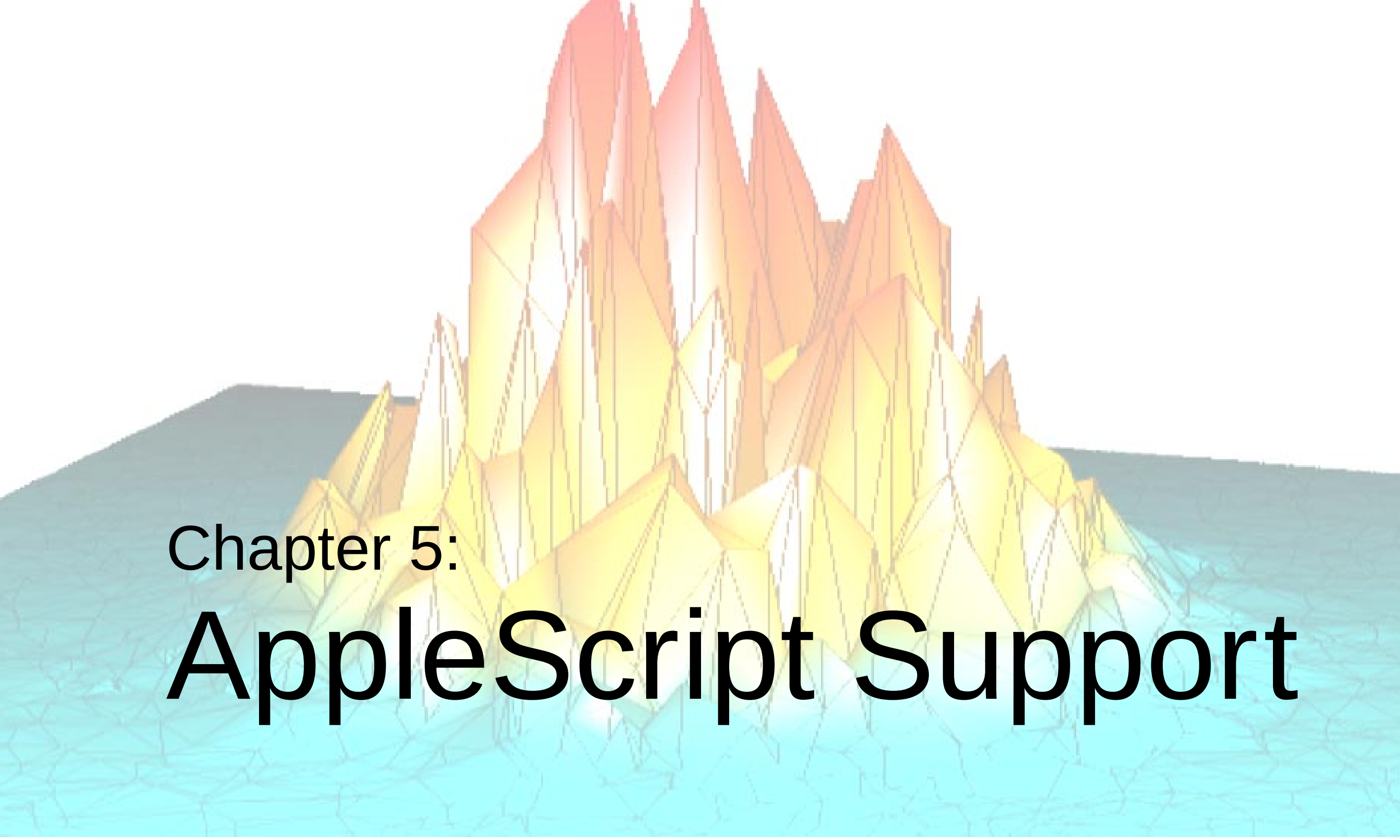
- MouseDown
- MouseMove
- MouseUp

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 5: AppleScript Support

This chapter describes the following topics:

AppleScript and IDL

IDL for Macintosh provides support for AppleScript, allowing IDL to control, and be controlled by, other applications running on the Macintosh. See Apple's AppleScript documentation for information on the basics of using AppleScript.

Procedures used when calling IDL for UNIX and IDL for VMS sharable objects are covered in "Using Callable IDL under Unix and VMS". Procedures used when calling the IDL DLL under Microsoft Windows are covered in "Using Callable IDL under Windows".

Basic AppleScript Support

IDL for Macintosh supports the four basic AppleScript commands that all applications are required to support. These allow you to launch and quit IDL, and to open and print documents.

Launching IDL

To launch IDL from AppleScript, use the AppleScript command:

```
tell application "IDL" to activate
```

Quitting IDL

To cause IDL to quit, use the AppleScript command:

```
tell application "IDL" to quit
```

Opening Documents

To have IDL open the text document `test.pro` on the disk `Macintosh HD`, use the AppleScript command:

```
tell application "IDL" to open file "Macintosh HD:test.pro"
```

Printing Documents

To have IDL print the text document `test.pro` on the disk `Macintosh HD` use the AppleScript command:

```
tell application "IDL" to print file "Macintosh HD:test.pro"
```

Using IDL Commands via AppleScript

IDL for Macintosh supports the AppleScript **do script** command. IDL responds to arbitrary text passed to it via AppleScript as if the text were entered at the IDL command prompt. Note that IDL must be waiting for input from the command line to accept the **do script** command from AppleScript.

For example, to instruct IDL to create an array of floating-point integers with the value of each element equal to its index and plot the array, use the following AppleScript command:

```
tell application "IDL" to do script "plot, findgen(20)"
```

Note

IDL executes commands it receives from AppleScript in the context of the currently active routine. Normally, this is the MAIN routine when IDL is waiting for input from the command line. If, however, IDL has stopped inside another routine (if a routine is active in the debugger, for example), the **do script** command will be executed in that routine's context.

Moving Data To and From IDL

IDL supports the AppleScript **get**, **set**, and **copy** commands to move data to and from IDL variables. To retrieve the value of a variable with the **get** or **copy** commands, the variable must already exist in the currently active routine. When setting the value of a variable with the **set** or **copy** commands, the variable will be created automatically in the currently active routine if it does not already exist.

For example, the following AppleScript command retrieves the value of the IDL variable “v” and saves it in the AppleScript variable **“v”**:

```
tell application "IDL" to get variable "v"
```

The following AppleScript command sets the value of the IDL variable “v” to the array [1, 2, 3]. If the IDL variable “v” does not exist, it is created.

```
tell application "IDL" to set variable "v" to {1, 2, 3}
```

The following AppleScript command copies the array [1, 2, 3] into the IDL variable “v”. If the IDL variable “v” does not exist, it is created.

```
tell application "IDL" to copy { 1, 2, 3 } to variable "v"
```

Notes

- IDL cannot get data from or move data to single- or double-precision complex variables or structure variables.
- In IDL, arrays are *indexed* in row-major format, meaning that the linear order of the references to data elements proceeds from the first element of the first row through the last element of the first row before beginning on the second row, and so on. AppleScript indexes arrays in column-major format, which means that the linear order of the references to data elements proceeds from the first element in the first column through the last element in the first column before beginning on the second column, and so on. Use the IDL routine TRANSPOSE to reverse the indexing order before passing the data using AppleScript.
- Passing large arrays using AppleScript is very inefficient, and is not recommended. In most cases it is more efficient to pass data in a file.
- Since it is not possible to determine what routine in IDL is executing from outside IDL, we recommend that you use the **get**, **set** and **copy** commands by executing an AppleScript script from *within IDL*, using the DO_APPLE_SCRIPT procedure.

Controlling Other Applications

IDL can execute arbitrary AppleScript scripts using the DO_APPLE_SCRIPT routine.

Importing Data into IDL

The following example shows how to use IDL to get data from a Microsoft Excel spreadsheet and plot it using IDL's surface command.

```
script = [ 'tell application "Microsoft Excel"', $
           'get value of range "R1C1:R5C5" of worksheet 1', $
           'end tell' ]
DO_APPLE_SCRIPT, script, RESULT = a
SURFACE, a
```

Exporting Data from IDL

The next example shows how to copy data from an IDL variable to Microsoft Excel. Once again, we work within IDL (using the DO_APPLE_SCRIPT procedure) so there is no confusion as to what routine context the variable comes from.

```
a = [ 1, 2, 3, 4, 5 ]
script = [ 'tell application "IDL" to copy variable "a" into t', $
   'tell application "Excel"', $
   'copy t to value of range "R1C1:R5C1" of worksheet 1', $
   'end tell' ]
DO_APPLE_SCRIPT, script
```

Controlling Other Applications

This example shows how to control the Metrowerks CodeWarrior application from within IDL. Note that a file name can be built in IDL and passed to another application using AppleScript.

First, use IDL to build a path name:

```
example = $
   FILEPATH('Call_Demo_PPC.proj', subdir = [ 'External', $
   'Examples', 'ShareLib' ])
```

Build an AppleScript script to run Metrowerks CodeWarrior:

```
script = [ $
  'with timeout of 600 seconds', $
  '  tell application "MW C/C++ PPC 1.2.1"', $
  '    activate', $
  '    open file "' + example + '"', $
  '    make project', $
```

```
      '    close project', $
      '    quit', $
      '  end tell', $
      'end timeout' ]
```

Execute the script:

```
   DO_APPLE_SCRIPT, script
```

IDL Apple Events

Application programs can communicate directly with IDL using Apple events. It is beyond the scope of this document to discuss how to write programs that use Apple events. For a discussion of this topic refer to *Inside Macintosh: Interapplication Communication* and *Apple Event Registry: Standard Suites*. This section discusses the three Apple events that IDL accepts.

Note

IDL understands the Apple events that support the **get**, **set**, **copy**, and **do script** AppleScript statements. In most cases, it is easier to use AppleScript as described above than to use the underlying Apple events. Note also that the **copy** AppleScript statement is implemented (by AppleScript) using the **Get Data** and **Set Data** Apple events. There is no **Copy Data** Apple event.

Do Script

The **Do Script** Apple event is used to ask IDL to perform actions specified in a script. IDL executes the specified text as if it were typed at the command line.

Event Class	kAEMiscStandards	
Event ID	kAEDoScript	
Parameters		
keyDirectObject		
	Description:	The script to execute
	Descriptor type:	typeChar
	Required or Optional:	Required
Reply Parameters		
keyErrorNumber		
	Description:	The error code
	Descriptor Type:	typeLongInteger
	Required or Optional:	Optional (Only returned if an error occurred)
keyErrorString		
	Description:	A character string describing the error
	Descriptor Type:	typeChar

	Required or Optional:	Optional (Only returned if an error occurred)
Notes	IDL only accepts the text version of this Apple event. It does not accept an alias record for specifying a file to execute.	
Result Codes		
	1	IDL Interpreter busy
	2	IDL statement too long

Get Data

The **Get Data** Apple event retrieves data from a specified IDL variable. Note that the specified variable must exist in the currently executing IDL routine, or in the MAIN context if no routine is executing.

Event Class	kAECoreSuite	
Event ID	kAEGetData	
Parameters		
keyDirectObject		
	Description:	The name of the variable
	Descriptor Type:	typeObjectSpecifier
	Required or Optional:	Required
keyAEKeyForm		
	Description:	The type of object specifier (must be of type formName)
	Descriptor Type:	typeEnumerated
	Required or Optional:	Required
keyAEKeyData		
	Description:	The actual variable name
	Descriptor Type:	typeChar
	Required or Optional:	Required
Reply Parameters		
keyAEResult		
	Description:	The variable's data
	Descriptor Type:	Either a single descriptor type or typeAEList

	Required or Optional:	Required
keyErrorNumber		
	Description:	The error code
	Descriptor Type:	typeLongInteger
	Required or Optional:	Optional (Only returned if an error occurred)
keyErrorString		
	Description:	A character string describing the error
	Descriptor Type:	typeChar
	Required or Optional:	Optional (Only returned if an error occurred)

Notes

IDL only accepts an object specifier record that specifies the variable by name.

IDL cannot return data from single- or double-precision complex variables or structure variables.

IDL indexes data in row-major format. See note under "Do Script" on page 98 for details.

Passing large arrays using AppleScript is very inefficient, and is not recommended. In most cases it is more efficient to pass data in a file.

Since it is not possible to determine what routine in IDL is executing from outside IDL, we recommend that you use the **get**, **set** and **copy** commands by executing an AppleScript script from *within IDL*, using the DO_APPLE_SCRIPT procedure. See "Controlling Other Applications" on page 96 for further information.

IDL does not support the optional parameter **keyAERequestedType ('rtyp')**. The returned data type is always the closest match to the IDL variable's data type.

Result Codes

errAECoercionFail (-1700)	Couldn't convert variable to AppleScript type
memFullErr (-108)	Not enough memory to get variable
-2	Variable not specified by name
-1	Invalid variable name
3	Variable undefined

Set Data

The **Set Data** Apple event copies data to a specified IDL variable in the currently executing routine, or in the MAIN context if no routine is executing.

Event Class	kAECoreSuite	
Event ID	kAEGetData	
Parameters		
keyDirectObject		
	Description:	The name of the variable
	Descriptor Type:	typeObjectSpecifier
	Required or Optional:	Required
keyAEKeyForm		
	Description:	The type of object specifier (must be of type formName)
	Descriptor Type:	typeEnumerated
	Required or Optional:	Required
keyAEKeyData		
	Description:	The actual variable name.
	Descriptor Type:	typeChar
	Required or Optional:	Required
keyAEData		
	Description:	The data to be copied into the variable
	Descriptor Type:	Either a single descriptor type or typeAEList
	Required or Optional:	Required
Reply Parameters		
keyErrorNumber		
	Description:	The error code
	Descriptor Type:	typeLongInteger
	Required or Optional:	Optional (Only returned if an error occurred)
keyErrorString		
	Description:	A character string describing the error

Descriptor Type:	typeChar
Required or Optional:	Optional (Only returned if an error occurred)

Notes

IDL only accepts an object specifier record that specifies the variable by name.

IDL cannot return data from single- or double-precision complex variables or structure variables.

IDL indexes data in row-major format. See note under "Do Script" on page 98 for details.

Passing large arrays using AppleScript is very inefficient, and is not recommended. In most cases it is more efficient to pass data in a file.

Since it is not possible to determine what routine in IDL is executing from outside IDL, we recommend that you use the **get**, **set** and **copy** commands by executing an AppleScript script from *within IDL*, using the DO_APPLE_SCRIPT procedure. See "Controlling Other Applications" on page 96 for further information.

Result Codes

errAECoercionFail (-1700)	Couldn't convert variable to AppleScript type
memFullErr (-108)	Not enough memory to get variable
-2	Variable not specified by name
-1	Invalid variable name
3	Couldn't create variable (variable undefined)

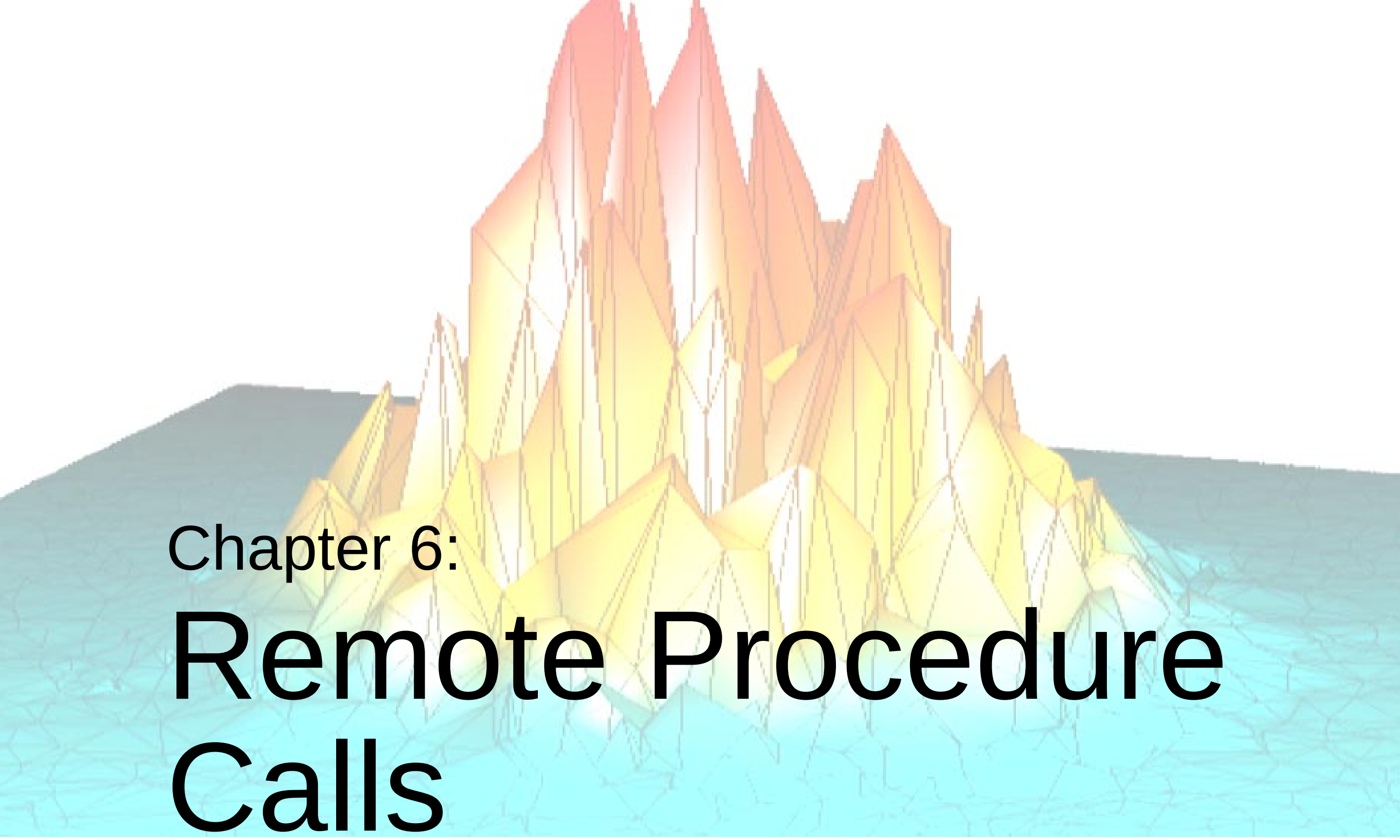
References

Schneider, Derrick. *The Tao of AppleScript*. Carmel, IN: Hayden Books, 1993. ISBN 1-56830-075-1

The following books, available from Apple Computer, may also be of interest:

Inside Macintosh: Interapplication Communication

Apple Event Registry: Standard Suites



Chapter 6:

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 111 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 `export.h` in the link 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

```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 6-1: 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 109. 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 109.

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:

`-2`	Illegal variable name (for example, "213xyz", "#a", "!DEVICE")
`-3`	Variable not transportable (for example, the variable is a structure or associated variable)

- **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 described below.

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 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 108.

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 108.

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 160.

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 160.

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 `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 108.

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 108.

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 160.

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 217.

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 216.

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 219.

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 218.

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_RPCVarIsArray(*v*)

This macro returns non-zero if *v* is an array variable.

IDL_RPCGetVarLong(*v*)

This macro returns the value of a 4-byte integer variable.

IDL_RPCGetVarString(*v*)

This macro returns the value of a string variable (as a *char**).

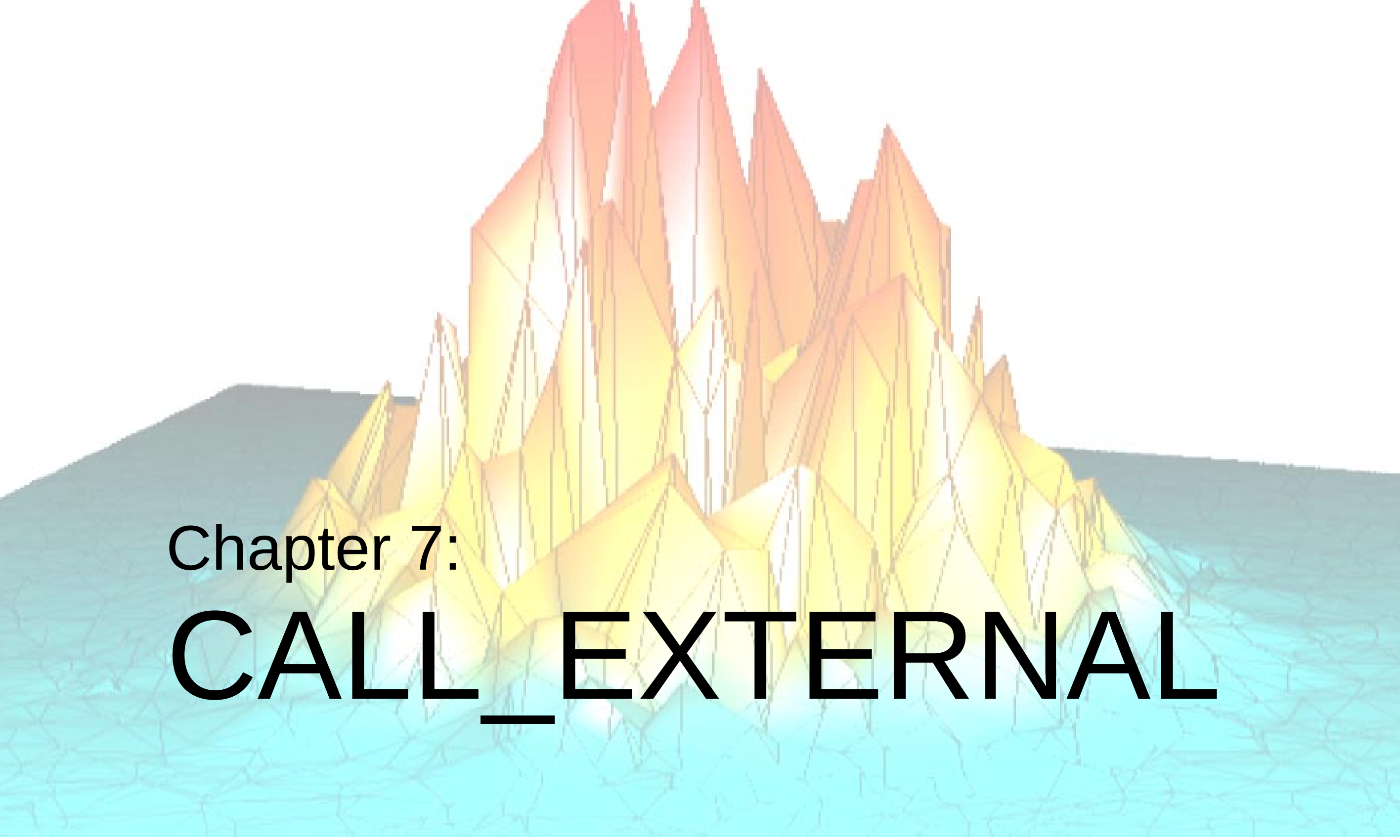
IDL_RPCGetVarType(*v*)

This macro returns the type code of the variable. IDL type codes are discussed in "Type Codes" on page 160.

RPC Examples

A number of example files are included in the *idldir*`/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 *idldir*/`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.



Chapter 7: CALL_EXTERNAL

This chapter discusses the following topics:

IDL and CALL_EXTERNAL

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 or Fortran, for example) from your IDL programs.
- 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 18, "Adding System Routines".

This chapter covers the basics of using CALL_EXTERNAL from IDL, then discusses platform-specific options for the UNIX, OpenVMS, Windows, and Macintosh versions of IDL.

The CALL_EXTERNAL Function

The CALL_EXTERNAL function loads and calls routines contained in shareable object libraries. 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 succeeding calls.

CALL_EXTERNAL is much easier to use than the LINKIMAGE routine. Unlike LINKIMAGE, 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.

For more information and examples, see one of the following sections:

- "CALL_EXTERNAL under UNIX" on page 147.
- "CALL_EXTERNAL under OpenVMS" on page 148.
- "CALL_EXTERNAL Under Windows" on page 156.
- "CALL_EXTERNAL on the Macintosh" on page 157.
- "CALL_EXTERNAL" in the *IDL Reference Guide*.

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()**. Using external code options for input and output, such as `stdin` or `stdout`, may generate unexpected results.

Memory Cleanup

IDL does not perform any memory cleanup calls on the values returned from the CALL_EXTERNAL routine. Because of this, any dynamic memory returned to IDL will not be returned to the system, which results 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. See "Dynamic Memory" on page 280 for more information.

Calling Convention and Parameter Passing

IDL calls routines in a shareable object using the C calling convention (`argc`, `argv`). Any routines called by CALL_EXTERNAL should be 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 may return. If this *return_type* is not IDL_LONG, a keyword must be used in the CALL_EXTERNAL call to indicate the type of the result.

The parameter `argc` is the count of optional parameters in the CALL_EXTERNAL call, and `argv` is an array of the parameters. Parameters are passed either by value or by reference. Parameters 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 a STRING structure in `argv[i]`. This structure is defined as follows:

```
typedef struct {
  unsigned short 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* for additional details about passing parameters by value. See the Fortran example below for a description of how to implement the (argc, argv) prototype in Fortran.

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];
```

The corresponding type in Fortran would be INTEGER*2.

Platform-Specific Information

For more information about calling conventions and parameter passing, see

- VMS: “Calling Convention and Parameter Passing” on page 148,
- Windows: “Calling Convention and Parameter Passing” on page 156.
- Macintosh:“Calling Convention and Parameter Passing” on page 157

Example: Passing Parameters by Reference to IDL

The routine in the following figure, `simple_vars.c`, accepts all of IDL's basic data types as parameters. The parameters are passed in by reference and the new squared values of the numbers are passed back to IDL.

C

```
#include <stdio.h>
#include "export.h"   /* IDL external definitions */

/* make sure that this routine is exported on the Macintosh */
#if defined(__POWERPC__)
__declspec(export) int simple_vars(int argc,void* argv[]);
#endif


int simple_vars(int argc,void* argv[])
{
     char *byte_var;
     short *short_var;
     /* IDL long variables don't map cleanly to a C type on
        all compilers. The IDL_LONG macro gives you a 32bit
        signed integer on all platforms. It is defined in export.h  */
     IDL_LONG *long_var;
     float *float_var;
     double *double_var;

     /* Ensure that the correct number of arguments were passed in */
     if(argc != 5) return 0;
     /* Cast the pointer in argv to the pointer variables */
     byte_var = (char *)   argv[0];
     short_var = (short *)  argv[1];
     long_var = (IDL_LONG *)   argv[2];
     float_var = (float *)  argv[3];
     double_var = (double *) argv[4];

     /* 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;
}
```

Figure 7-1: Passing Parameters by Reference to IDL — simple_vars.c

You can call `simple_vars.c` from IDL using the following statements:

```
B=2B & I=3 & L=3L & F=0.0 & D=0.0D
R = CALL_EXTERNAL('simple_vars.so','simple_vars',B,I,L,F,D)
```

Example: Calling a C routine

You can add a routine that returns the sum of a floating point array, similar to the TOTAL function in IDL. The `example.c` routine is shown in the following figure.

C

```c
#include <stdio.h>
float sum_array(argc, argv)
int argc;
void *argv[];
{
  float *fp, s = 0.0; int n;
  for(n = *(int *) argv[1], fp = (float *) argv[0]; n--; )
    s += *fp++;
  return(s);
}
```

Figure 7-2: Calling a C routine — example.c

You can use the following statements to compile and link `example.c` to produce a shareable object library for the Solaris operating system. For more information on compiling and linking, see "UNIX Compilation and Linking" on page 147.

```
cc -I RSI-Directory/IDL-Directory/external -c -kpic example.c
cc -G -o example.so example.o
```

where *`RSI-directory`* is the name of the main installation directory and *`IDL-Directory`* indicates the version of IDL installed (for example. `idl_5.3`).

The compiled routine resides in the shared library `example.so`, so it can be called by the following IDL code.

Note

Under some operating systems, an underscore character must be added before or after the function name in order to match the entry point name in the object file. The entry point name is generated by the compiler according to the rules of the system linker, and may be different for different operating systems or compilers.

```
;Make an array.
X = FINDGEN(10)
S = CALL_EXTERNAL('example.so', $
   'sum_array' X, N_ELEMENTS(X), /F_VALUE)
```

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 `F_VALUE` keyword specifies that the returned value is a floating-point number rather than an IDL_LONG.

Example: Calling a Fortran Routine Using a C Interface Routine

Calling Fortran is similar to calling C, with the restriction 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.

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, i.e. it expects all arguments to be addresses. If C passes a pointer (an address) by value, Fortran will interpret it correctly as an 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.

As in the above example, we assume that the routines are being compiled under Solaris. The object name of the Fortran subroutine will be `sum_array1_` to match the output of the Solaris Fortran compiler. The contents of `example_c2f.c` and `example.f` are shown in the following figures:

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 */
}
```

Figure 7-3: C Wrapper Used to Call Fortran Code (example_c2f.c)

C

```
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
```

Figure 7-4: 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 *`RSI-Directory`*`/IDL-Directory/external/fortran`. 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.`

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 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 restriction 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.

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 different 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 *`RSI-Directory`*`/IDL-Directory/external/fortran`*.* This directory also contains a makefile, which builds this example on UNIX platforms.

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
```

Figure 7-5: 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.

Further Examples

A number of example routines, along with makefiles for the various systems that support CALL_EXTERNAL, are located in the `call_external` subdirectory of the `external` subdirectory of the IDL distribution. The README file in that directory contains instructions for building the examples.

Wrapper routines

CALL_EXTERNAL routines are often very sensitive to the arguments they receive. In many cases, 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 often good to have a wrapper routine (written in IDL) to ensure that the arguments that are passed to the external code are always correct. The IDL procedure shown in the following figure is one example of a wrapper for the `simple_vars` routine.

IDL

```
PRO simple_vars,b,i,l,f,d,DEBUG=debug
   if NOT(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 apropriate
   ;type, which may cause errors to be thrown if structures are passed in.
   b = (SIZE(b,/TNAME) EQ 'UNDEFINED') ? 2b :  byte(b[0])
   i = (SIZE(i,/TNAME) EQ 'UNDEFINED') ? 3  :  fix(i[0])
   l = (SIZE(l,/TNAME) EQ 'UNDEFINED') ? 4L :  long(l[0])
   f = (SIZE(f,/TNAME) EQ 'UNDEFINED') ? 5.0 : float(f[0])
   d = (SIZE(d,/TNAME) EQ 'UNDEFINED') ? 6.0D :  double(d[0])

   PRINT,'Calling simple_vars with the following arguments:'
   HELP,b,i,l,f,d
   IF (CALL_EXTERNAL(lib_name('call_examples'),'simple_vars',/PORTABLE,$
                     b,i,l,f,d) EQ 1) then BEGIN
       PRINT,'After calling simple_vars:'
       HELP,b,i,l,f,d

   ENDIF ELSE MESSAGE,'External call to simple_vars failed'
END
```

Figure 7-6: 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

The `lib_name` function (line 16 in the preceding figure) returns its argument with the proper shareable library suffix for the target platform (for example:`.a`, `.dll`, `.so`, `.sl`)

The following table lists the IDL supported data types and the corresponding C data types:

IDL	C
BYTE	char or unsigned char
INT	short
UINT	unsigned short
LONG	IDL_LONG
ULONG	IDL_ULONG
LONG64	IDL_LONG64
ULONG64	IDL_ULONG64
FLOAT	float
DOUBLE	double
STRING	IDL_STRING
STRUCT	C structure with the same layout

Table 7-1: Data type mapping between IDL and C

Handling Different Data Types

This section describes how to convert complex IDL data types such as strings to C data types, and how to use C data types in IDL.

Strings

IDL represents strings internally as IDL_STRING descriptors. For more information about IDL_STRING, see Chapter 9, "IDL Internals: Variables" and Chapter 11, "IDL Internals: String Processing". These descriptors are defined in the C language as:

```
typedef struct {
  unsigned short 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.

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 figure is an example that 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.

C

```
#include <stdio.h>
#include "export.h"
/* IDL_STRING is declared like this:
   typedef struct {
      unsigned short slen;           Length of string, 0 for null
      short stype;                   type of string, static or dynamic
      char *s;                       Addr of string
   } IDL_STRING;

However, you should rely on the definition in export.h instead
of declaring your own string structure.
*/
#include <string.h>

/* make sure that this routine is exported on the Macintosh */
#if defined(__POWERPC__)
__declspec(export) char* string_array(int argc,void* argv[]);
#endif


char* string_array(int argc,void* argv[])
{
     IDL_STRING*str_descr;
     IDL_LONG n;/* number of elements in array*/

     int max_index; /* index of longest string */
     int max_sofar; /* length of longest string*/
     int i;
/* IDL will make a copy of the string that is returned (if it is not NULL).
So, to avoid a memory leak the return value should be a pointer to a stat-
ic buffer containing a null terminated string. */
#define MAX_OUT_LEN 511 /*any string longer than this will be truncated*/
static char result[MAX_OUT_LEN +1];  /*leave a space for a ’\0’ on the
longest string */
/* make sure there are the correct  # of arguments.
      IDL will convert the NULL into an empty string (’’). */
   if (argc != 2) return((char *)NULL);

   /*  Cast the pointers in argv to local variables. */
   str_descr= (IDL_STRING *) argv[0];
   n= *(int*) argv[1];
```

C

```

/*  Check the size of the array passed in. n should be > 0.*/
   if (n < 1) return (char*)NULL;
   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;
       }
   }
/* 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*) NULL;
   /* copy the longest string into the buffer. Since result was declared
      static, it will initially be filled with zeros. And, since the buffer
      is 1 byte longer than MAX_OUT_LEN, the last byte of the buffer should
      already be a ’\0’.  This is important, because if the input string
      to strncpy() is longer than MAX_OUT_LEN, strcpy() will _not_ write
      a ’\0’.*/
   strncpy(result,str_descr[max_index].s,MAX_OUT_LEN);
   return(result);
#undef MAX_OUT_LEN
}
```

Figure 7-7: Handling String Variables in External Code — string_array.c

Arrays

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 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 program shown in the following figure calculates the sum of a subsection of a two dimensional array:

C

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

/* make sure that this routine is exported on the Macintosh */
#if defined(__POWERPC__)
__declspec(export) double sum_2d_array(int argc,void* argv[]);
#endif

double sum_2d_array(int argc,void* argv[])
{
     /* 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. */
     double* arr;
     IDL_LONG x_start,x_end,x_size,y_start,y_end,y_size,x,y;

     double result = 0.0;

     if (argc != 7) return 0.0;

     arr = (double*)argv[0];
     x_start = *(int*)argv[1];
     x_end = *(int*)argv[2];
     x_size = *(int*)argv[3];
     y_start = *(int*)argv[4];
     y_end = *(int*)argv[5];
     y_size = *(int*)argv[6];

     /* 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 export.h */
     x_start = IDL_MAX(x_start,0);
     y_start = IDL_MAX(x_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;
}
```

Figure 7-8: Adding the Elements of a 2D IDL Array — sum_2d_array.c

The System Routine interface provides much more support for the manipulation of IDL array variables. See Chapter 18, “Adding System Routines” for more information.

Structs

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.,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 figure increments each field of an IDL structure of type ASTRUCTURE:

C

```c
#include <stdio.h>
#include "export.h"
/*
**  C definiton 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;

/* make sure that this routine is exported on the Macintosh */
#if defined(__POWERPC__)
__declspec(export) int incr_struct(int argc,void* argv[]);
#endif

int incr_struct(int argc, void *argv[])
{
    ASTRUCTURE* mystructure;
    IDL_LONG n;

    int i;
    if (argc != 2) return 0;
    mystructure = (ASTRUCTURE*) argv[0]; /* 1st arg is structure array */
    n = *(int*) argv[1];              /* 2nd arg is number of elements */

    /* for each structure in the array, increment every field */
    for (i = 0;i<n;i++,mystructure++)
    {
       mystructure->zero++;
       mystructure->one++;
       mystructure->two++;
       mystructure->three++;
       mystructure->four[0]++;
       mystructure->four[1]++;
    }
    return 1;
```

Figure 7-9: Accessing an IDL Structure from a C Routine — incr_struct.c

It is not possible to access structures with unknown layouts or nested structures using the CALL_EXTERNAL interface. The System Routine interface does provide support for determining the layout of a structure at runtime and for accessing nested structures. See “CALL_EXTERNAL” in the *IDL Reference Guide* for more information.

CALL_EXTERNAL under UNIX

The CALL_EXTERNAL function loads and calls routines contained in shareable object libraries.

An important advantage to calling external routines with CALL_EXTERNAL, as opposed to spawning child processes and passing parameters by pipe, is that IDL and the called routine share the same memory and data space. CALL_EXTERNAL avoids the overhead of process creation and parameter passing. In addition, the shareable object file containing the called routine is only loaded the first time it is referenced.

UNIX Compilation and Linking

Each UNIX system has different compilation and link statements for producing a shareable object suitable for usage with CALL_EXTERNAL. Also, the name of the entry point in the object may be different, because compilers may add leading or trailing underscores to the name of the source routine.

Compilation and linking statements for the UNIX platforms supported by IDL are collected in the file `callext_unix.txt` in the `call_external` subdirectory of the `external` subdirectory of the main IDL directory. The statements in this file can be used to for the example routines above. They also show the correct flags to use for any set of C routines. Additional libraries may be added to the link lines by using the `-L` and `-l` flags, except on certain systems noted below, where the name of the library must be specified explicitly.

CALL_EXTERNAL under OpenVMS

By default under VMS, the CALL_EXTERNAL function loads and calls routines contained in shareable images that adhere to the OpenVMS calling standard using the LIB$CALLG() runtime library function. It is also possible to use the portable convention available on the other platforms (discussed in "CALL_EXTERNAL under UNIX" on page 147) by specifying the PORTABLE keyword to CALL_EXTERNAL.

Alpha/OpenVMS Restrictions

IDL uses the `LIB$CALLG()` function to implement CALL_EXTERNAL under OpenVMS. The ALPHA/OpenVMS procedure calling specification states that certain floating values are passed in certain registers and not on the stack as with the VAX. `LIB$CALLG()` cannot put these arguments into the correct location because its interface does not tell it the types of the arguments it is passing. Accordingly, under Alpha/OpenVMS, CALL_EXTERNAL is restricted in the following ways:

- A single- or double-precision floating-point argument can only be passed by value if it is one of the first six arguments to the function.
- Single- and double-precision complex arguments cannot be passed by value.

Calling Convention and Parameter Passing

IDL calls routines in the shareable image object with the parameters that were passed to CALL_EXTERNAL. Unlike UNIX routines that are called using the C calling convention `argc` and `argv`, OpenVMS routines are called with a parameter list. Any routine called by CALL_EXTERNAL should be defined in the following manner:

In IDL:

```
status = CALL_EXTERNAL(image, 'example', p1, p2, p3, p4 )
```

In C:

```
return_type example( p1, p2, p3, p4 )
```

where *`return_type`* is one of the types which CALL_EXTERNAL may return. If this type is not IDL_LONG, then a keyword must be used in the CALL_EXTERNAL call to indicate the type of the result.

Arrays are passed by reference. By default, scalars are passed by reference also. See "CALL_EXTERNAL" in the *IDL Reference Guide* for additional details about passing parameters by value.

It is important to note that IDL integer variables correspond to a C short integer. For example, an integer variable could be defined in an IDL routine as follows:

```
IDL> A=5
```

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;
```

The declaration in a called Fortran routine should be:

```
INTEGER*2 A
```

Example: Calling a C routine

Assume you wish to add a routine that returns the sum of a floating-point array, similar to the TOTAL function in IDL. This routine accepts a pointer to the array (`f_arr`), a pointer to the number of elements (`n_ele`), and it passes back a floating point sum as a parameter (`sum`). It does not pass back a return value. Even though the return value is unused, the CALL_EXTERNAL call in IDL must be made as a function call. In this case, the return value should be assigned to a dummy value.

The following figure shows the contents of the `example.c` file. See "OpenVMS Compilation and Linking" on page 155 for details on compiling and linking the code on your platform.

C

```
#include <stdio.h>

void sum_array(float *f_arr, long *n_ele, float *sum)
{
  float s;

  for(s=0.0; *n_ele--; ) /* Compute the sum */
  s += *f_arr++;

  *sum = s;
}
```

Figure 7-10: Returning the Sum of a Floating-Point Array — sum_array

Assuming the compiled routine resides in the executable file `example.exe`, it can be called with the following IDL code.

```
;Make an array.
X = FINDGEN(10)
```

```
;Define a variable to hold the result.
sum = 0.0
S = CALL_EXTERNAL('[path]example.exe', $
   'example' X, N_ELEMENTS(X), sum)
```

In this example, `[path]example.exe` is the full path name of the linked executable file, *`example`* is the name of the entry point, and *`X`*, *`N_ELEMENTS(X)`*, and *`sum`* are passed to the called routine as parameters. Note that the result is returned in the variable *`sum`*, which must be defined as the proper data type (single-precision floating-point in this case) before calling the external routine. You may find it helpful to replace `[path]example.exe` with an OpenVMS logical name. For example:

```
$ DEFINE IDL_EXAMPLE [path]example.exe
```

Example: Calling a Fortran Routine

Calling Fortran is similar to calling C, with the restriction 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.

To compute the sum of a float array using Fortran you would use the Fortran subroutine such as that shown in the following figure (file `example.f`). See "OpenVMS Compilation and Linking" on page 155 for details on compiling and linking the code on your platform.

f77

```
SUBROUTINE EXAMPLE( F_ARR, N_ELE, SUM)

INTEGER*4N_ELE

REAL*4F_ARR(N_ELE)
REAL*4SUM

INTEGER I

SUM = 0.0

DO I=1, N_ELE
SUM = SUM + F_ARR(I)
END DO

RETURN
END
```

Figure 7-11: Fortran Version of sum_array()

Example: Calling the TPU Editor

A simple example uses CALL_EXTERNAL to invoke the TPU text editor directly from IDL. The TPU editor is provided as a shareable, callable image in the file `SYS$SHARE:TPUSHR.EXE`. Its entry point is named `TPU$EDIT`. The statement

```
status = CALL_EXTERNAL('tpushr', 'tpu$edit', 'test.pro', '')
```

calls `TPU` to edit the file `test.pro`. `TPU$EDIT` requires two parameters: the file to be edited and an output file. In this case, the output file is a zero-length string denoting that the output file is the same as the input.

A procedure named EDIT is easily written to accept a filename parameter, verify that it is a scalar string, and call TPU:

```
;Edit file using TPU.
PRO EDIT, file

;Is the parameter defined? Print error message.
IF N_ELEMENTS(file) EQ 0 THEN BEGIN
  BAD_PAR:  PRINT, "Usage:  EDIT, 'filename'"

RETURN
ENDIF
;Validate parameter.
S = SIZE(file)

;Is the string scalar?
IF (S(0) NE 0) OR (S(1) NE 7) THEN GOTO, BAD_PAR

;Call the editor.
STATUS = CALL_EXTERNAL('tpushr', 'tpu$edit', 'file', '')
END
```

Example: Calling a Runtime Library Function

This example presents a procedure called GETMSG that returns the error message text given an OpenVMS error message number.

The *`OpenVMS Run-Time Library`* routine `LIB$SYS_GETMSG` is used. Briefly, parameters to `LIB$SYS_GETMSG` are:

```
LIB$SYS_GETMSG(MSG_ID, MSG_LEN, DEST_STRING, FLAGS)
```

where

- `MSG_ID` is the message identification code, a longword passed by reference.

- MSG_LEN is the returned message length, a longword integer passed by reference.
- DEST_STRING is the string into which the message is placed, passed by reference to its string descriptor.
- FLAGS is a longword integer, passed by reference, of flag bits that determine message content (default value 0).

The code of the procedure GETMSG is as follows:

```
;Return the text to the OpenVMS message MSGID.
FUNCTION GETMSG, msgid

;Destination string length, initialize to longword.
len = 0L

;Destination string, initialize to a string containing 100 blanks.
msg = STRING(REPLICATE(32B, 100))

;Call lib$sys_getmsg; flags parameter is 0.
istat = CALL_EXTERNAL('librtl', 'lib$sys_getmsg', $
   LONG(msgid), len, msg, 0L)

;Truncate the string using the returned length.
msg = STRMID(msg, 0, len)

;Return the string.
RETURN, msg
END
```

This example illustrates one method of returning a string from an external routine. The function creates a 100-character, blank-filled string called MSG, which is passed by reference to its descriptor to LIB$SYS_GETMSG. The descriptor LIB$SYS_GETMSG fills the memory pointed to by this descriptor with the result, which can be up to 100-characters long, and returns the actual string length in the variable len. The call to STRMID is necessary to truncate the string MSG to the length of the returned string.

Calling a VMS Fortran Subroutine

This example calls a simple VMS Fortran function that returns the mean of a floating-point array:

```
MEAN = CALL_EXTERNAL(/F_VALUE, 'MY_AVG_EXE', 'MY_AVG', $
   FINDGEN(10), 10L)
```

The F_VALUE keyword indicates that the called function returns a single-precision, floating value.

The entry MY_AVG, inside a shareable image pointed to by the logical name MY_AVG_EXE, is called. The two parameters passed to the routine are a 10-element, floating-point vector and the element count, passed as a long integer. The Fortran routine, in file MY_AVG.FOR, is shown in the following figure.

f77

```
REAL * 4 FUNCTION MY_AVG(V, N)
REAL*4 V(*)
MY_AVG = 0.0
DO I=1,N
  MY_AVG = MY_AVG + V(I)
END DO
MY_AVG = MY_AVG / N
RETURN
END
```

Figure 7-12: Returning the Mean of a Floating-Point Array Using Fortran— my_avg.for

The OpenVMS DCL commands to compile, link, and setup the logical name are as follows:

```
$ FORTRAN MY_AVG
$ LINK MY_AVG, SYS$INPUT/OPT/SHARE
UNIVERSAL = MY_AVG
$ DEFINE MY_AVG_EXE dduu:[xxx]MY_AVG.EXE
```

The OpenVMS Alpha DCL commands to compile, link, and setup the logical name are as follows:

```
$ FORTRAN MY_AVG
$ LINK MY_AVG, SYS$INPUT/OPT/SHARE
SYMBOL_VECTOR = [MY_AVG=PROCEDURE]
$ DEFINE MY_AVG_EXE dduu:[xxx]MY_AVG.EXE
```

Passing Parameters by Value

Scalar parameters can be passed by value or by reference. The optional `VALUE` keyword parameter is a byte array in which nonzero elements indicate the respective scalar parameter is to be passed by value. Parameters are passed by reference if the `VALUE` parameter is not present or the respective element is zero.

For example, if the above routine, `MY_AVG`, is written in the C language and declared as follows:

```
float my_avg(float *, int)
```

The call from IDL becomes:

```
MEAN = CALL_EXTERNAL(/F VALUE, VALUE = [0B,1B], $
   'MY_AVG_EXE', 'MY_AVG', FINDGEN(10), 10L)
```

causing the second parameter, the number of elements, to be passed by `value` rather than by reference.

Note

Under Alpha/OpenVMS:

A single- or double-precision floating-point argument can only be passed by value if it is one of the first six arguments to the function.

Single- and double-precision complex arguments cannot be passed by value.

See "Alpha/OpenVMS Restrictions" on page 148 for a more detailed discussion of these restrictions.

Using CALL_EXTERNAL with Fortran Common Blocks

In Fortran language routines, common blocks are declared as writeable, shareable PSECTS. This can cause the following error messages:

```
LIB-E-ACT: error activating image: image.exe
SYSTEM-F-NOTINSTALL: writeable shareable images must be installed
%CALL_EXTERNAL: error in called routine
```

If this occurs, place a line like the following in the linker options file for each common block:

```
PSECT_ATTR = com_block_name, noshr
```

where *`com_block_name`* is the name of the common block in your Fortran routine. These lines must be added for each common block in the routines that you call using CALL_EXTERNAL.

Further Examples

A number of example routines, along with makefiles, are located in the `call_external` subdirectory of the `external` subdirectory of the IDL distribution. To compile the examples into a shareable object and then run IDL procedures, see the README files located in either *`RSI-Directory/`*`IDL-Directory/external/C` or *`RSI-Directory/`*`IDL-Directory/external/fortran`.

OpenVMS Compilation and Linking

Compilation and linking statements for OpenVMS are collected in the file `callext_vms.txt` in the `call_external` subdirectory of the `external` subdirectory of the main IDL directory. The statements in this file can be used for the example routines above.

CALL_EXTERNAL Under Windows

You can use CALL_EXTERNAL to call your own Win32 Windows-compatible DLLs.

There are some difficulties in creating Windows-compatible DLLs and we suggest that you obtain some additional information on this topic (such as Charles Petzold's book *Programming Windows 95*) before attempting to write one.

Calling Convention and Parameter Passing

All DLLs called with CALL_EXTERNAL must use an `argc`-`argv` calling convention described in "CALL_EXTERNAL under UNIX" on page 147. The parameters you specify in the call to CALL_EXTERNAL are translated into the `argv` vector.

Parameters can be passed by value or by reference. See "CALL_EXTERNAL" in the *IDL Reference Guide* for additional details about specifying how parameters are to be passed.

DLL procedures can return long integers, floating-point integers, or strings by value. They can return any sort of information by reference.

Examples

Example DLL code, a makefile, and an IDL procedure calling the newly-created DLL are located in the `call_external` subdirectory of the `external` subdirectory of the IDL distribution. To compile the example into a DLL, issue the following command:

```
C:\RSI\IDL\EXTERNAL\CALL_EXTERNAL> nmake /f makefile_win.mak
```

The makefile uses the correct options for the Microsoft C compiler. If you use a different compiler, you may need to change the compilation or link flags. See the makefile for details on the compiler used.

CALL_EXTERNAL on the Macintosh

You can use CALL_EXTERNAL to call your own Macintosh shared library files.

Calling Convention and Parameter Passing

IDL calls routines in a shared library using the C calling convention (`argc`, `argv`) described in "CALL_EXTERNAL under UNIX" on page 147. Any routines called by CALL_EXTERNAL should be defined with a prototype similar to the following:

```
return_type resFunction(int argc, void *argv[])
```

where *`return_type`* is one of the data types which CALL_EXTERNAL may return. If this *`return_type`* is not IDL_LONG, a keyword must be used in the CALL_EXTERNAL call to indicate the type of the result.

When you build a shared library, you must tell the linker which symbols are exported. You can use any of the following methods:

1. Use a __declspec(export) declaration in your code. This is the easiest method if the number of exported symbols is small, and is the method used in this manual. See the example in the following figure for an example of this approach.
2. Supply an export file to the linker. This is the best option if the number of symbols is large.
3. Specify that the linker export all symbols.

The details of how to use the linker depend on which development environment you are using (MPW, Code Warrior, and so on) and are not discussed in this manual. Consult your system documentation for details.

Example: Calling a C Routine on a PowerPC Macintosh

CALL_EXTERNAL on the PowerMac expects native PowerPC code to be stored in the data fork of a shared library file. It loads the code with the routine **GetDiskFragment()**, and finds the requested routine using **FindSymbol()**. Because of this, you may put multiple routines in the same shared library file and even share global variables amongst them. The library is only loaded once, and is freed when IDL exits.

Consult your compiler manual on how to create a shared library file. Consult the Code Fragment Manager of *Inside Macintosh: PowerPC System Software* to learn about **GetDiskFragment()** and **FindSymbol()**.

This example is a routine that returns the sum of an integer array (similar to the TOTAL function in IDL) accepts a variable length argument list where the first parameter is the number of arguments and the second is an array of pointers to the arguments. The function then returns a long integer which is the sum of its inputs.

The `example.c` file contains the code shown in the following figure. You must link the program as a shared library. In Metrowerks CodeWarrior, this is done using a popup menu in the “Project” section of the “Preferences” dialog.

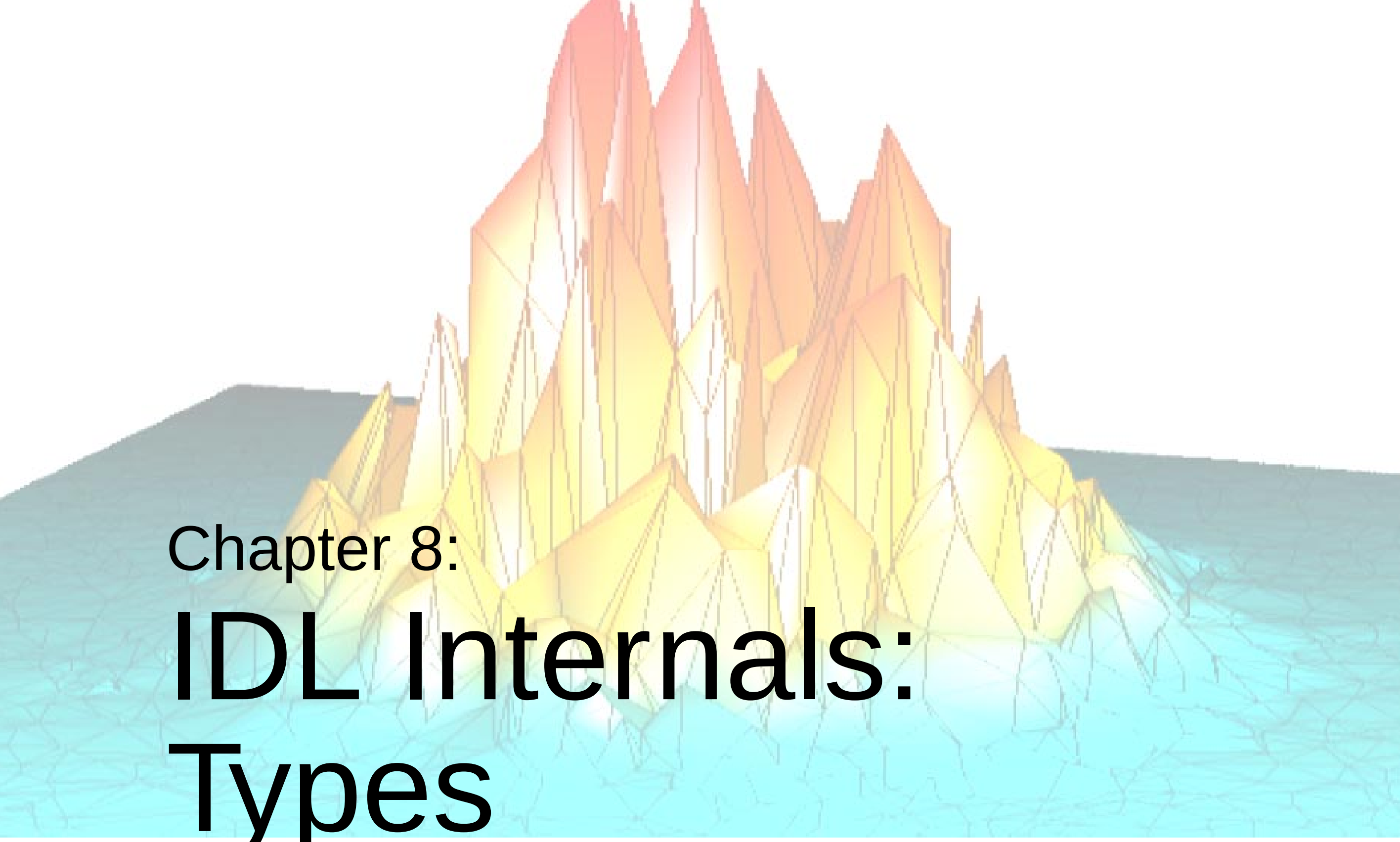
C

```
#if defined(__POWERPC__)
__declspec(export) long sumarray(int argc, void *argv[]);
#endif

long sumarray(int argc, void *argv[])
{
  longretval;
  short*arrInd;
  inti;

  retval = 0L;

  if (argc == 2) {
    /*
     * IDL integer arrays are arrays of shorts so get the pointer
     * to the array (arrays are always passed to CALL_EXTERNAL by
     * reference
     */
    arrInd = (short *) argv[0];

    /*
     * then just sum the array
     */
    for (i = 0; i < *(int *)argv[1]; i++)
      retval += *arrInd++;
  }

  return(retval);

}            /* end of summer routine */
```

Figure 7-13: Returning the Sum of an Integer Array — sumarray

To use the above example in IDL, enter:

```
IDL> testarr = INDGEN(10)
IDL> total = CALL_EXTERNAL("example", "sumarray", testarr, 10L)
```



Chapter 8: 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 unlikely, 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 `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	short
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 175
IDL_TYP_DCOMPLEX	Double precision complex	IDL_DCOMPLEX
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 8-1: IDL Types and Mapping to C

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 `export.h`.

Unsigned Integer Data

IDL_UINT represents the unsigned 16-bit data type and is defined in `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 `export.h` is used instead.

Unsigned Long Integer Data

IDL_ULONG represents the unsigned 32-bit data type and is defined in `export.h`.

64-bit Integer Data

IDL_LONG64 represents the 64-bit data type and is defined in `export.h`.

Unsigned 64-bit Integer Data

IDL_ULONG64 represents the unsigned 64-bit data type and is defined in `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 {
  unsigned short 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 280 and “Getting Dynamic Memory” on page 188.

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 213.

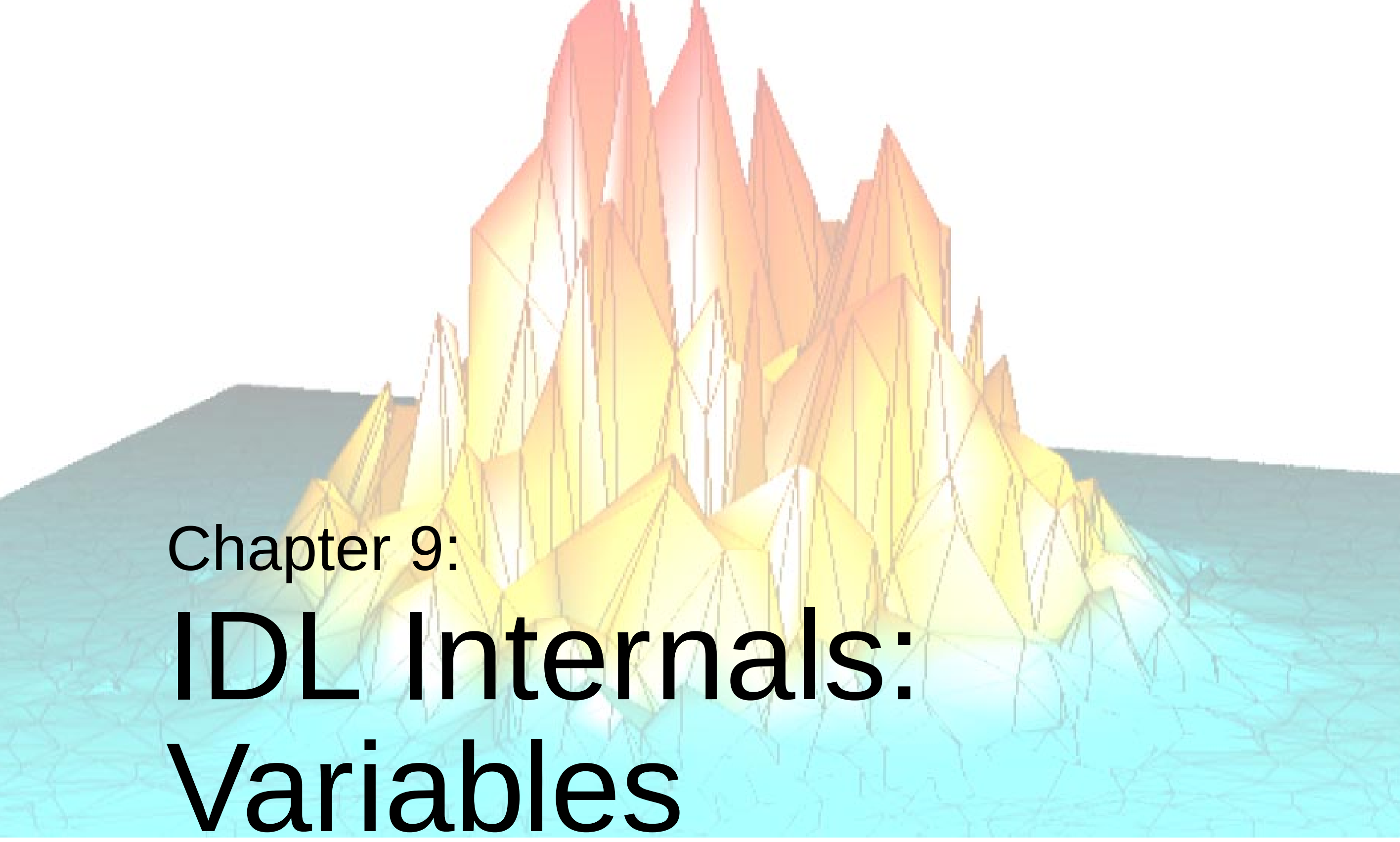
IDL_MEMINT and IDL_FILEINT Types

Most of the IDL-supported operating systems limit memory and file lengths to a signed 32-bit integer (approximately 2.3 GB). These limitations may change dramatically in the future: some systems have 64-bit memory capabilities and others support files longer than 2^{31}-1 bytes. 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. Use these special types to ease the evolutionary move to larger memory and files.

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 163. IDL is currently limited to 32-bit signed (2^{31}-1) bytes of memory, meaning that IDL_TYP_MEMINT is currently mapped to IDL_TYP_LONG. On some systems, IDL allows access to files larger than 32-bits; IDL_TYP_FILEINT is mapped to IDL_TYP_LONG64. On other systems, IDL_TYP_FILEINT is mapped to IDL_TYP_LONG.

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 them in the appropriate places without special interpretation. These types can be used anywhere that a normal IDL type can be used, such as in keyword processing.

Programmers should be aware of the IDL_MEMINTScalar() and IDL_FILEINTScalar() functions, described in "Converting Arguments to C Scalars" on page 234.



Chapter 9: 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.

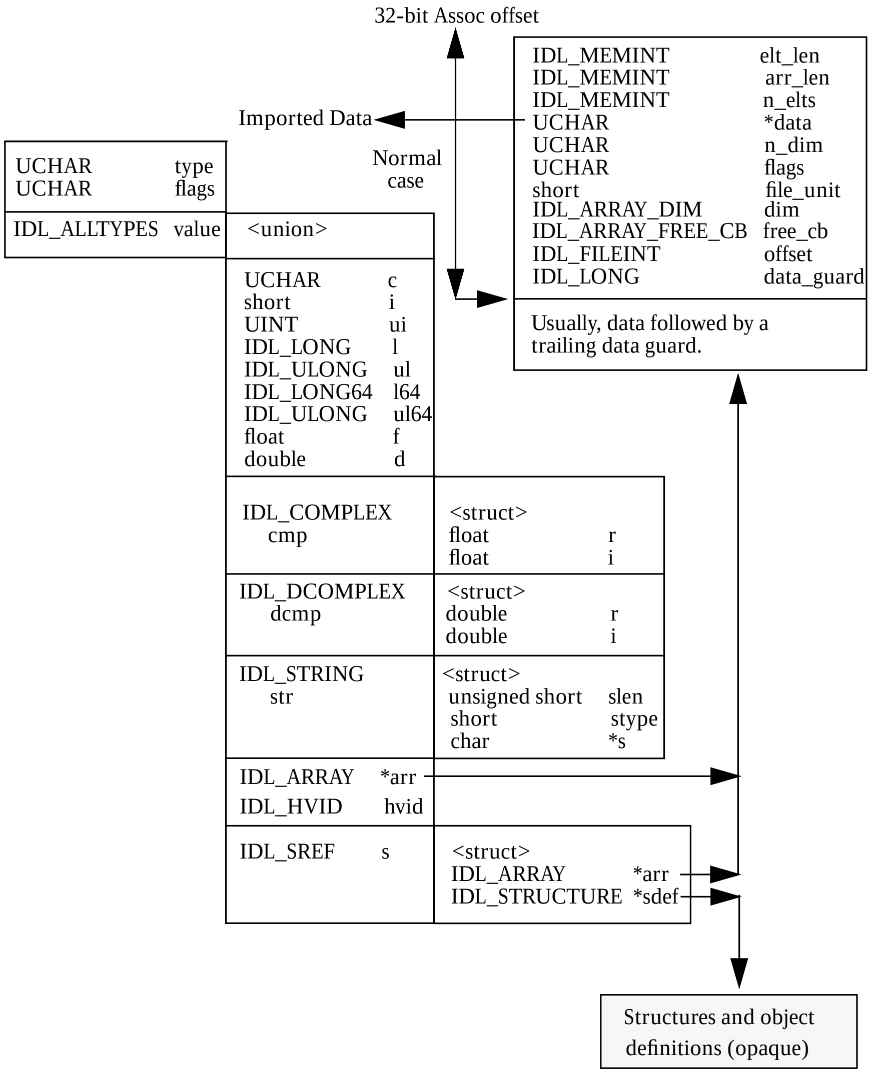


Figure 9-1: Structure of an IDL variable

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 */
  short 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 160. 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 231). 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 181.

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 thirteen basic data types (IDL structures are never scalar). 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

Table 9-1: 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 struct {
  IDL_MEMINT elt_len;
  IDL_MEMINT arr_len;
  IDL_MEMINT n_elts;
  char *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:

```
short *data;        /* Declare a pointer variable */
data = (short *) 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. Currently, only one bit value is defined for this field:

> 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.

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 173. It is worth noting that in the definition of the **IDL_ALLTYPES** union (described on page 169), 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 Research Systems 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 186.

IDL_STRUCT_TAG_DEF is defined as:

```
typedef struct {
  char *name;
  IDL_LONG *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 cast to void (e.g., **(void *) IDL_TYP_BYTE**).

flags

This field is reserved to RSI, and must be set to 0.

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_LONG one = 1;
static IDL_LONG tag2_dims[] = { 3, 2, 3, 4};
static IDL_LONG 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 [2] [3] [4];
          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_LONG IDL_StructTagInfoByName(IDL_StructDefPtr sdef, char
*name,
        int msg_action, IDL_VPTR *var)
IDL_LONG 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 the interpreter is exiting because an error was detected, allocated temporaries are expected, and are reclaimed quietly. Hence, your routines need only return temporaries on normal return, not before issuing errors. See "IDL Internals: Error Handling" on page 221.

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.

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 160.

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.

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 160.

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 blind 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 280). 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 returned 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 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 160.

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,
                         IDL_ALLTYPES *value)
```

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 160".

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 221.

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()**.

Note that new variables can only be created at the MAIN level. Make sure that no user procedures or functions are active when you call these function.

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.

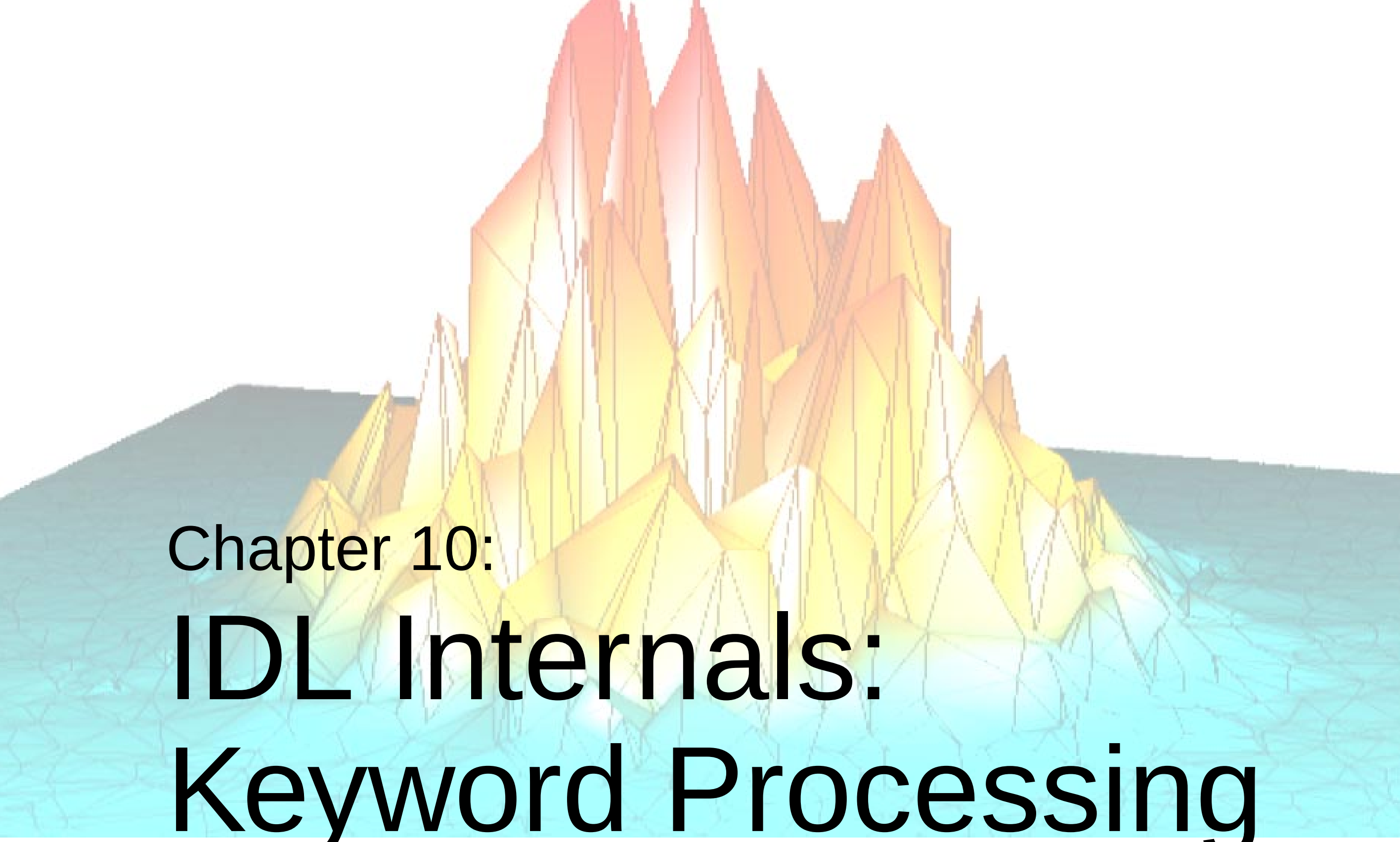
ienter

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 constrain.



Chapter 10: 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 295, you must register your system routine before IDL will recognize it. When registering the routine, you indicate that it accepts keyword arguments by OR-ing the constant **IDL_SYSFUN_DEF_KEYWORDS** into the **flags** field of the **IDL_SYSFUN_DEF2** struct passed to **IDL_SysRtnAdd()**, or by setting the KEYWORDS keyword in a call to LINKIMAGE.

Routines defined in this way must be designed to handle 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.

The function **IDL_KWGetParams()** 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. Finally, the **IDL_KWCleanup()** function is used in certain circumstances to clean up.

More information about these routines and structures can be found in the following sections.

The IDL_KW_PAR Structure

The **IDL_KW_PAR** struct provides the basic specification for keyword processing. The **IDL_KWGetParams()** 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_KWGetParams()** 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_KWGetParams() automatically converts the keywords to a specified type. This field specifies the desired type code. For scalars, the only allowable types are **IDL_TYP_LONG, IDL_TYP_FLOAT, IDL_TYP_DOUBLE**, and **IDL_TYP_STRING**. Any type can be specified for arrays, or for no conversion, **IDL_TYP_UNDEF** (0).

mask

The enable mask. This field is ANDed with the mask argument to **IDL_KWGetParams()** 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.

For example, 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.

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 the **IDL_VPTR** pointed to 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 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** pointed to 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 passed by reference. The address of the **IDL_VARIABLE** or expression is stored in the value field as with **IDL_KW_OUT**. Expressions and/or constants are valid. If **IDL_KW_VIN** is specified, **IDL_KWCleanup()** must be called with a **IDL_KW_MARK** parameter before **IDL_KWGetParams()** is called. **IDL_KWCleanup()** must be called with a **IDL_KW_CLEAN** parameter before your routine exits to properly return temporary variables that may have been allocated by **IDL_KWGetParams()**.

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, the low 12 bits of this field (**flags**) will be bitwise ORed with the longword pointed to by the **value** field. Note that this implies the **IDL_TYP_LONG** type code, and is incompatible with the **IDL_KW_ARRAY** and **IDL_KW_OUT** flags.

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 203. 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

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;
  int nmin;
  int nmax;
  int 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()**.

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 C variable specified by the **value** field.

Array Input-Only

Array input-only keywords work similarly to the scalar case, except that the **value** field points to an **IDL_KW_ARR_DESC** 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. 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** struct, and if no error results, the number is written into the **n** field of that struct.

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_OUT** or **IDL_KW_VIN** flag is set in the **IDL_KW_PAR** struct. In this case, you receive the **IDL_VPTR** to the actual keyword argument, and 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.

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 "The IDL_KW_PAR Structure" on page 200) 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 "Don't allow the macros used in the above switch statement to remain defined beyond the scope of this function." on page 316.

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 <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   };
```

C

```

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 215.

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.

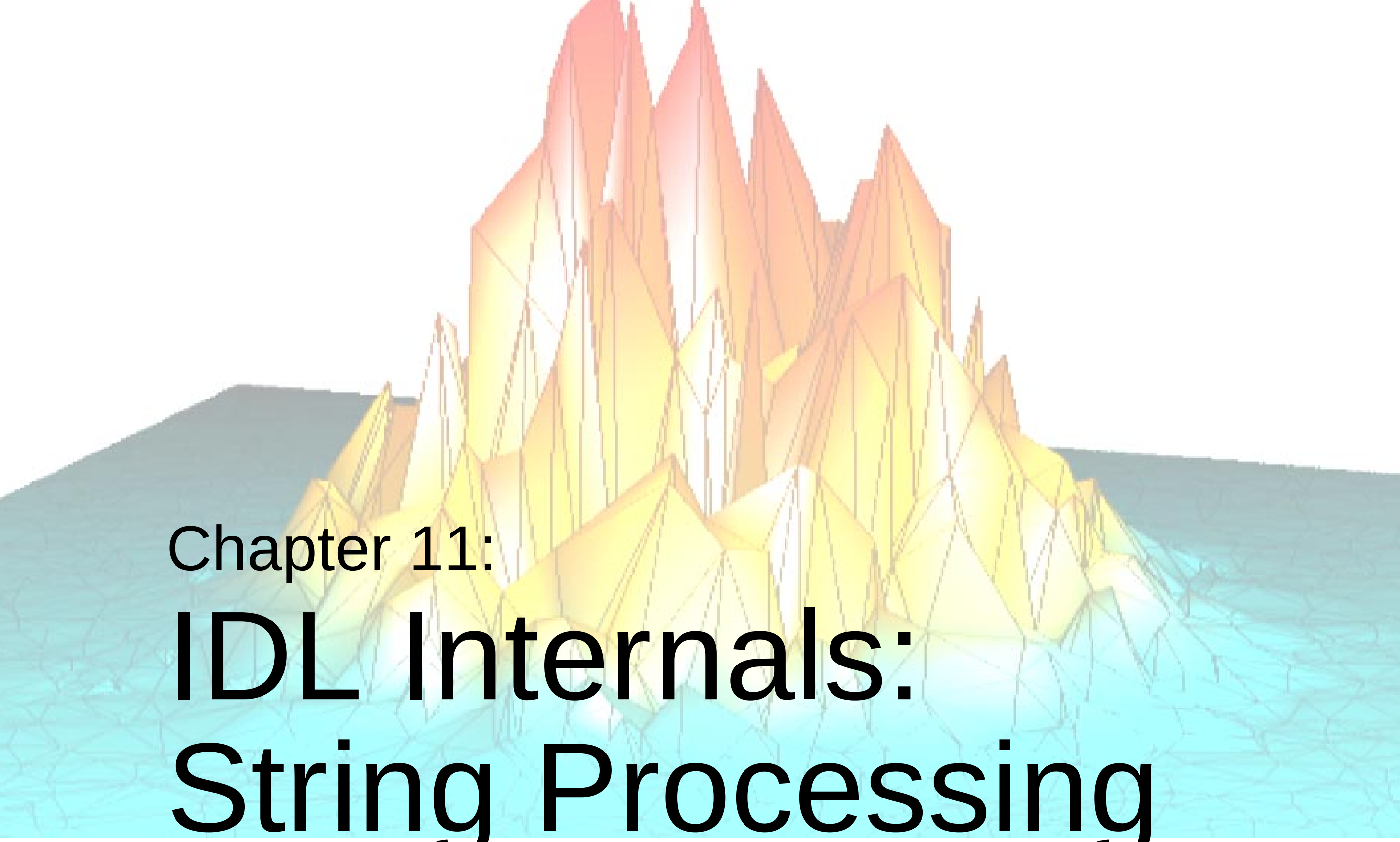
47 – 55

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 256. 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**.



Chapter 11: 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.

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:

```
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 12: 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 331 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 I**DL_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 Research Systems 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 Research Systems 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 the **IDL_Message()** or **IDL_MessageFromBlock()** functions. These functions are patterned after the standard C library **printf()** function.

```
void IDL_Message(int code, int action, ...)
void IDL_MessageFromBlock(IDL_MSG_BLOCK block,int code,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 that are available to programmers adding system routines to IDL. The use of these codes is described below. See "IDL_M_GENERIC" on page 227 and "IDL_M_NAMED_GENERIC" on page 227.

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 !ERR, !ERROR, and !ERR_STRING 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.

IDL_MSG_ATTR_SYS

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. The OpenVMS Standard C Library (stdio) also provides this variable. 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**. Note that the OpenVMS stdio contains emulations of the Unix system calls in addition to the functions normally found in the Unix stdio. 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. Under OpenVMS, this applies to those functions that emulate the Unix system calls. Otherwise, **errno** might contain an unrelated garbage value resulting in an incorrect error message.

The Macintosh and Microsoft Windows operating systems have **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.

...

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, Research Systems 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

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.
```

Specifying errno Explicitly

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 226 for additional information about **errno** and its use.
2. The **IDL_MSG_ATTR_SYS** modifier code for the action argument is ignored.

Issuing OpenVMS Messages

The **IDL_Message()** function is used when issuing general errors, or issuing Unix-specific errors. This includes those errors reported by standard C library functions under any operating system, as such libraries generally emulate the Unix functionality reasonably well. However, **IDL_Message()** is not adequate for reporting errors that come from OpenVMS system routines (System Services and Run-Time Library). Therefore, OpenVMS-specific errors are reported using the **IDL_MessageVMS()** or **IDL_MessageVMSFromBlock()** function.

The **IDL_MessageVMS()** function is only available under OpenVMS, and should only be used in code that can’t work under other operating systems due to system dependencies. In cases where either would work, always use **IDL_Message()**.

IDL_MessageVMS() is very similar to **IDL_Message()**:

```
void IDL_MessageVMS(int code, int err1, int err2, int action, …)
void IDL_MessageVMSFromBlock(IDL_MSG_BLOCK block, int code, int
err1, int err2, int action, …)
```

The arguments are identical to those for **IDL_Message()** (see “Issuing Error Messages” on page 224) with the following exceptions:

err1

IDL_MessageVMS() always issues a single line error message that describes the problem from the IDL point of view. The **err1** argument is used to specify that a second line containing a system error message should also be issued, allowing the user to get a complete picture of what went wrong. If this argument is 0, no system error is issued. Otherwise, it should be set to the OpenVMS status code for the failed operation.

err2

Some OpenVMS system routines return 2 error codes. For example, RMS often does this. If there are 2 error codes, **err2** should be used to report the second one. If this argument is 0, no second system error is issued.

Looking Up A Message Code by Name

Given a message block pointer and the name of a message from that block, the I**DL_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 192) 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 302.

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

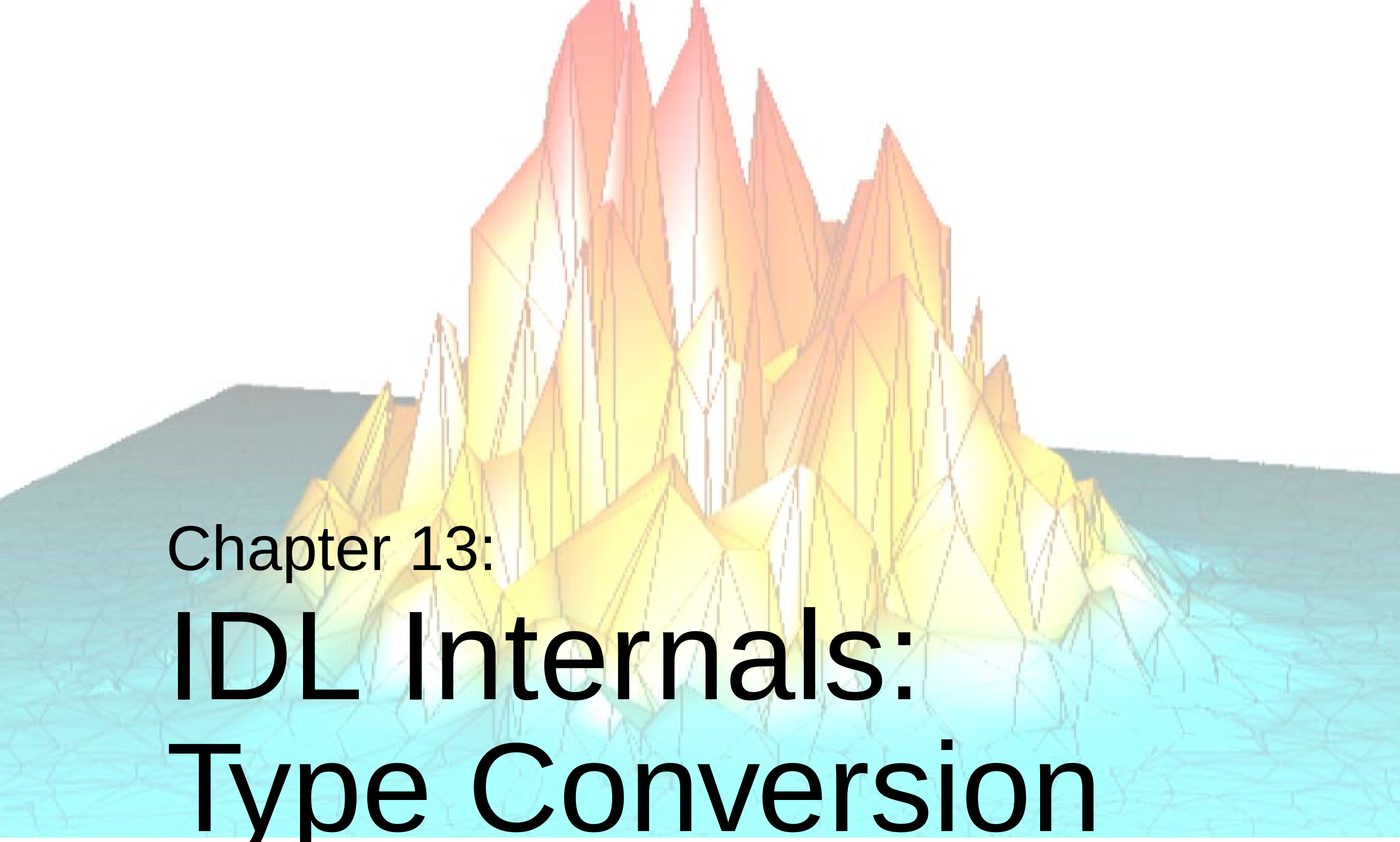
The argument must be **IDL_TYP_STRING**.

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 13: IDL Internals: Type Conversion

This chapter discusses the following topics:

Converting Arguments to C Scalars

The **IDL_LongScalar()** and **IDL_DoubleScalar()** functions convert the value of their argument to a C scalar. In addition, **IDL_MEMINTScalar()** and **IDL_FILEINTScalar()** exist for processing memory and file sizes.The converted value is returned as the function value. The functions are defined as:

```
IDL_LONG IDL_LongScalar(IDL_VPTR p)
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 231. **IDL_DoubleScalar()** works exactly like **IDL_LongScalar()** except that it returns a double-precision, floating-point value.

For example, the following IDL system function (named PRINT_LONG) prints the value of its first argument, converted to long 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 160.

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.

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

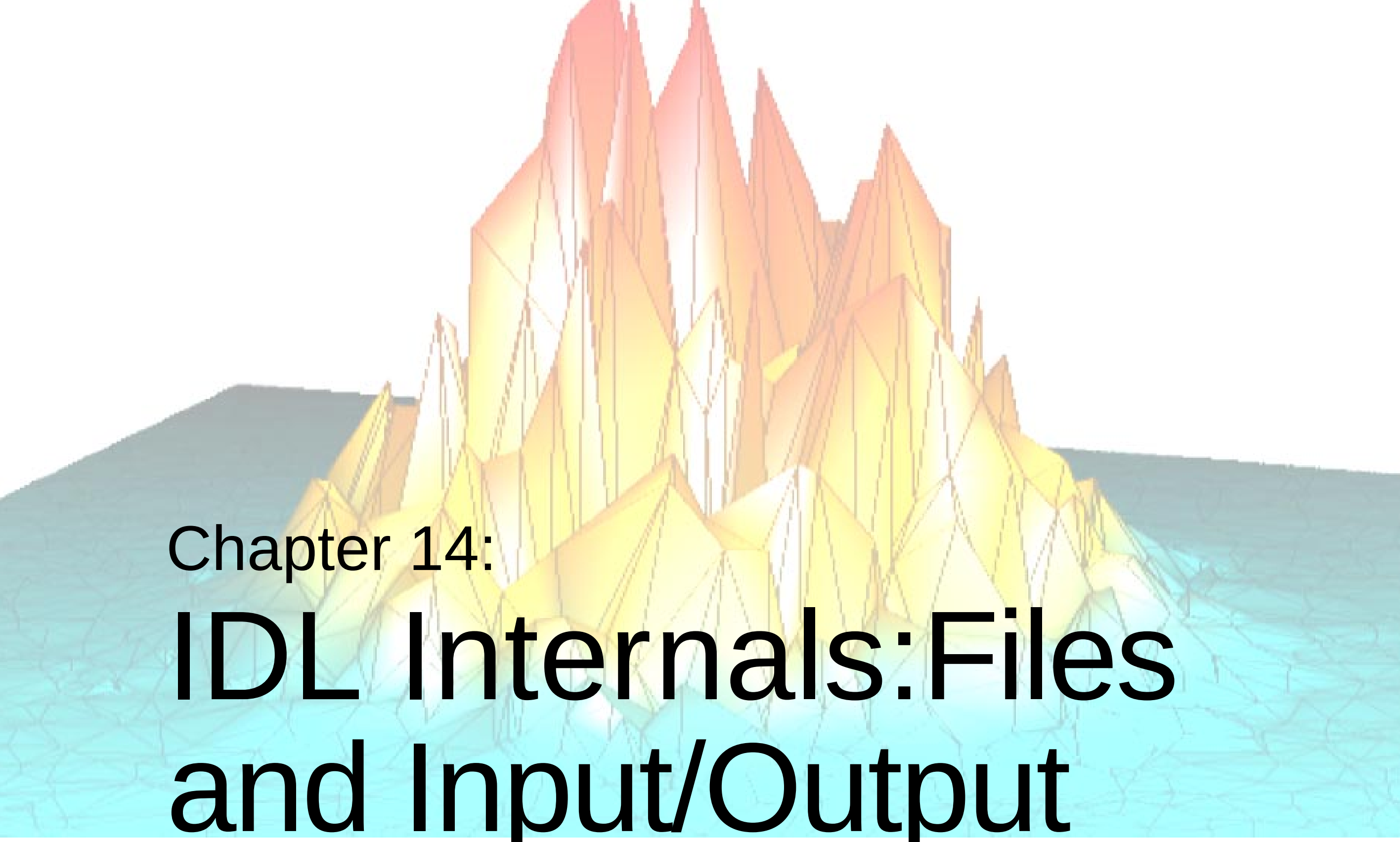
A series of functions exist to convert **VARIABLE**s to specific types:

```
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 14:

IDL Internals:Files and Input/Output

This chapter discusses the following topics:

IDL and Input/Output files

On most platforms supported by IDL, file handling is built on the standard C library stream package. For OpenVMS this is true only in the case of stream files—RMS is used in all other cases.

Most system routines should not do Input/Output directly. It is almost always better to write a function or procedure that returns a result, which a user can then print in any format supported by the IDL I/O subsystem. For this reason, only minimal I/O abilities are available. Most stream file abilities are present, but access to RMS files is only marginally supported.

If your application *must* perform I/O, it is best to use stream files. Using stream files gives your application the best chance of working with all operating systems supported by IDL. 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 14-1: Disallowed C Library Routines and their IDL counterparts.

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 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_LONG64 flags;
  FILE *fptr;
  struct {
    unsigned short mrs;
  } rms;
} IDL_FILE_STAT;
```

Warning

In IDL versions prior to IDL 5.3, the flags field of the IDL_FILE_STAT struct was a 32-bit number. In IDL 5.3, flags has been widened to a 64-bit number. All user code that calls the IDL_FileStat() function must be recompiled.

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 14-2: 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.
IDL_F_SR	The file is a SAVE/RESTORE file.

Table 14-3: Bit values for the flags field

Bit Value	Description
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 stdio buffering will be performed for the file and all data transfers will go directly to the operating system for processing. 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 value, but both values 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.
IDL_F_VMS_FIXED	The file has fixed-length records.
IDL_F_VMS_VARIABLE	The file has variable-length records.
IDL_F_VMS_SEGMENTED	The file has VMS Fortran segmented records.

Table 14-3: Bit values for the flags field

Bit Value	Description
IDL_F_VMS_STREAM	The file is treated as a VMS stream file, opened via the standard C library, just like under UNIX. VMS attempts to convert non-stream files into a logical stream in order to mask the fact that the file is not really a stream file.
IDL_F_VMS_STREAM_STRICT	The file is treated as a VMS stream file, opened via the standard C library, just like under UNIX. In the case of non-stream files, no attempt is made to convert the file contents to a logical stream.
IDL_F_VMS_RMSBLK	The file is open for RMS block mode access. New files are created with fixed-length 512-byte records.
IDL_F_VMS_RMSBLKUDF	The file is open for RMS block mode access. New files are created with the UNDEFINED record type. One result of this is that most VMS utilities won't be able to read this file.
IDL_F_VMS_INDEXED	The file has indexed organization.
IDL_F_VMS_PRINT	The file will be sent to the VMS system printer SYS$PRINT when it is closed.
IDL_F_VMS_SUBMIT	The file will be sent to the standard VMS system batch queue SYS$BATCH when it is closed.
IDL_F_VMS_TRCLOSE	When closed, the file allocation will be truncated to the amount actually used.
IDL_F_VMS_CCLIST	The file has carriage return carriage control.
IDL_F_VMS_CCFORTRAN	The file has Fortran-style carriage control.
IDL_F_VMS_CCNONE	The file data contains explicit carriage control.
IDL_F_VMS_SHARED	Shared access to the file is allowed.

Table 14-3: Bit values for the flags field

Bit Value	Description
IDL_F_VMS_SUPERCEDE	Supersede existing version on open.
IDL_F_DOS_BINARY	The file is in binary mode.

Table 14-3: Bit values for the flags field

fptr

The stream file pointer to the file. This field can be used with standard library functions to perform I/O. This field is always valid under non-VMS operating systems 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. Under VMS, the stream file pointer is only valid if the file is open for stream access. In this case the IDL_F_VMS_STREAM bit will be set in the flags field.

rms.mrs

For RMS (VMS) record oriented files, this field contains the record length.

Opening Files

Files are opened using the IDL_FileOpen() function.

Warning

In IDL versions prior to IDL 5.3, the extra_flags argument to the IDL_FileOpen() function was a 32-bit number. In IDL 5.3, extra_flags has been widened to a 64-bit number. All user code that calls the IDL_FileOpen() function must be recompiled.

IDL_FileOpen()

```
int IDL_FileOpen(int argc, IDL_VPTR *argv, char *argk, int
   access_mode, IDL_LONG64 extra_flags, int longjmp_safe)
```

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.
IDL_OPEN_TRUNC	The file was truncated when it was opened. This implies that IDL_OPEN_W is also set.

Bit Value	Description
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 14-4: Bit values for the access_mode argument

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 239). Note that it makes no sense to set the IDL_F_ISATTY bit in this mask.

Warning

In IDL versions prior to IDL 5.3, the extra_flags argument to the IDL_FileOpen() function was a 32-bit number. In IDL 5.3, extra_flags has been widened to a 64-bit number. All user code that calls the IDL_FileOpen() function must be recompiled.

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.

Special File Units

There are three files that are always open. Under VMS, these units are open as both stream and RMS files—each unit is opened twice, once as a stream and again as an RMS variable length record file. This means that you can always refer to the fptr field for these units without checking the IDL_F_VMS_STREAM bit of the flags field. These are the only three units for which this is true. Finally, the constant IDL_NON_UNIT always has a value which is not a valid file unit. 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.

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 file descriptor (see "File Information" on page 239). 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.

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.

There are two macros provided to make preventing/enabling of this bit easy:

- **IDL_FILE_NOCLOSE(unit)** — Given the file LUN, this macro sets the IDL_F_NOCLOSE bit.
- **IDL_FILE_CLOSE(unit)** — Given the file LUN this macro clears the IDL_F_NOCLOSE bit.

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.

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 224). 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 14-5: 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, and IDL_EFS_NOXDR, in addition to other operating system specific concerns.
IDL_EFS_NOT_NOSTDIO	The file was not opened with IDL_F_UNIX_NOSTDIO attribute under UNIX.

Table 14-5: Bit values for the flags argument

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 = TYP LONG;
unit.flags = 0;
IDL_FileGetUnit(1, argv);

/* Set up the file name */
name.type = TYP STRING;
name.flags = 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, IDL_F_VMS_STREAM,
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 the discussion in the *IDL Reference Guide* for additional details.

Adding to the Journal File

IDL_Logit()

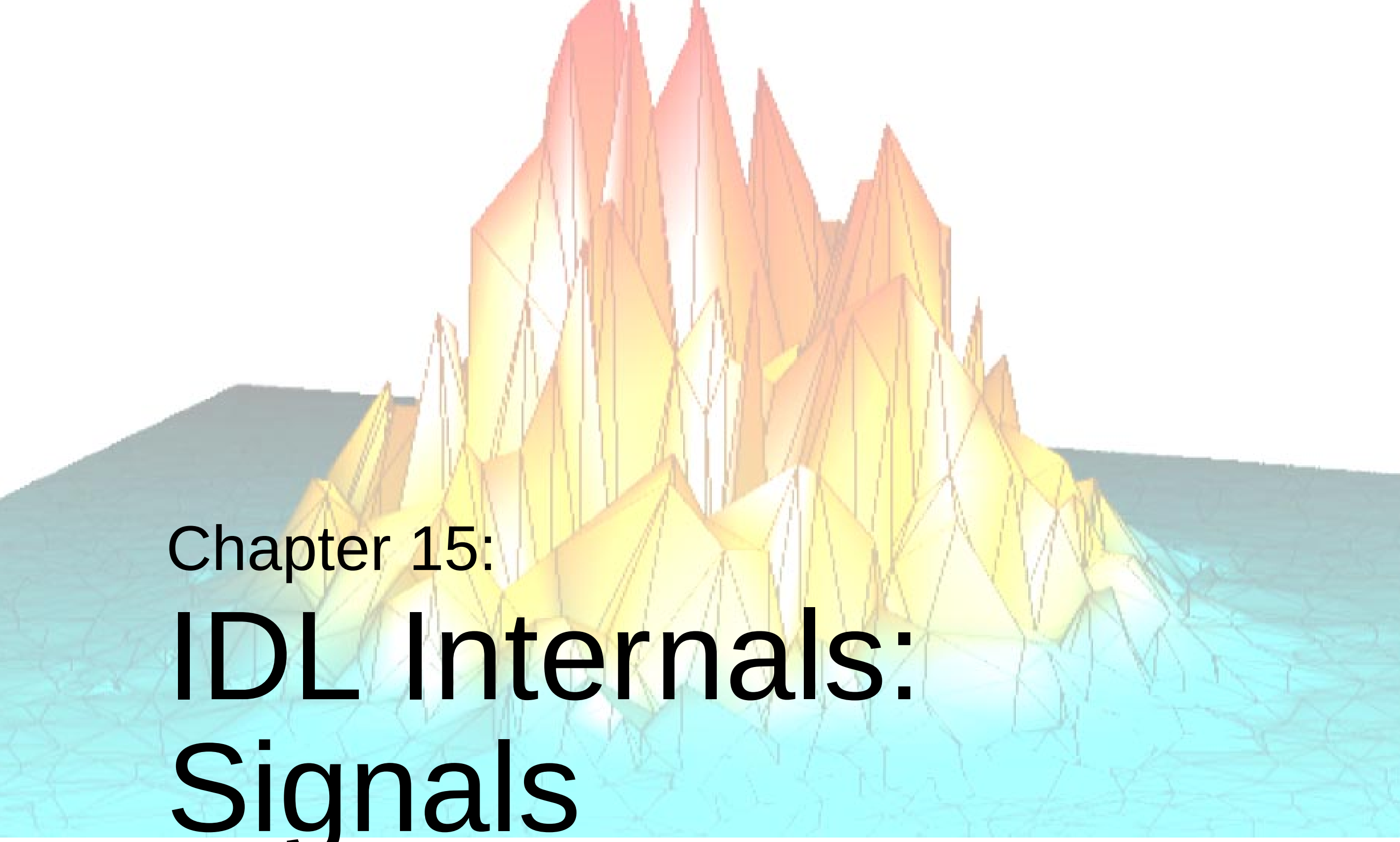
The IDL_Logit() function can be used to add lines of output to the journal log file:

```
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 15: IDL Internals: 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. 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. Although this discussion refers primarily to UNIX IDL, signals *are* used in minimal ways under other operating systems supported by IDL.

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. Also, some platforms that do support Posix signals fail to provide needed information for **SIGFPE** and **SIGTRAP**, which are very important to IDL's exception handling.
- 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 using CALL_EXTERNAL, LINKIMAGE, or Callable IDL chooses a library 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.
- Most systems provide extra information for **SIGFPE** and **SIGTRAP** that allow the program to deduce what type of arithmetic problem occurred and continue execution. The format of this information differs widely. The details of continuing execution are highly OS- and hardware-dependent, and are often undocumented.

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 constant set of definitions and routines, and by handling details like re-establishing handlers.
- 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()

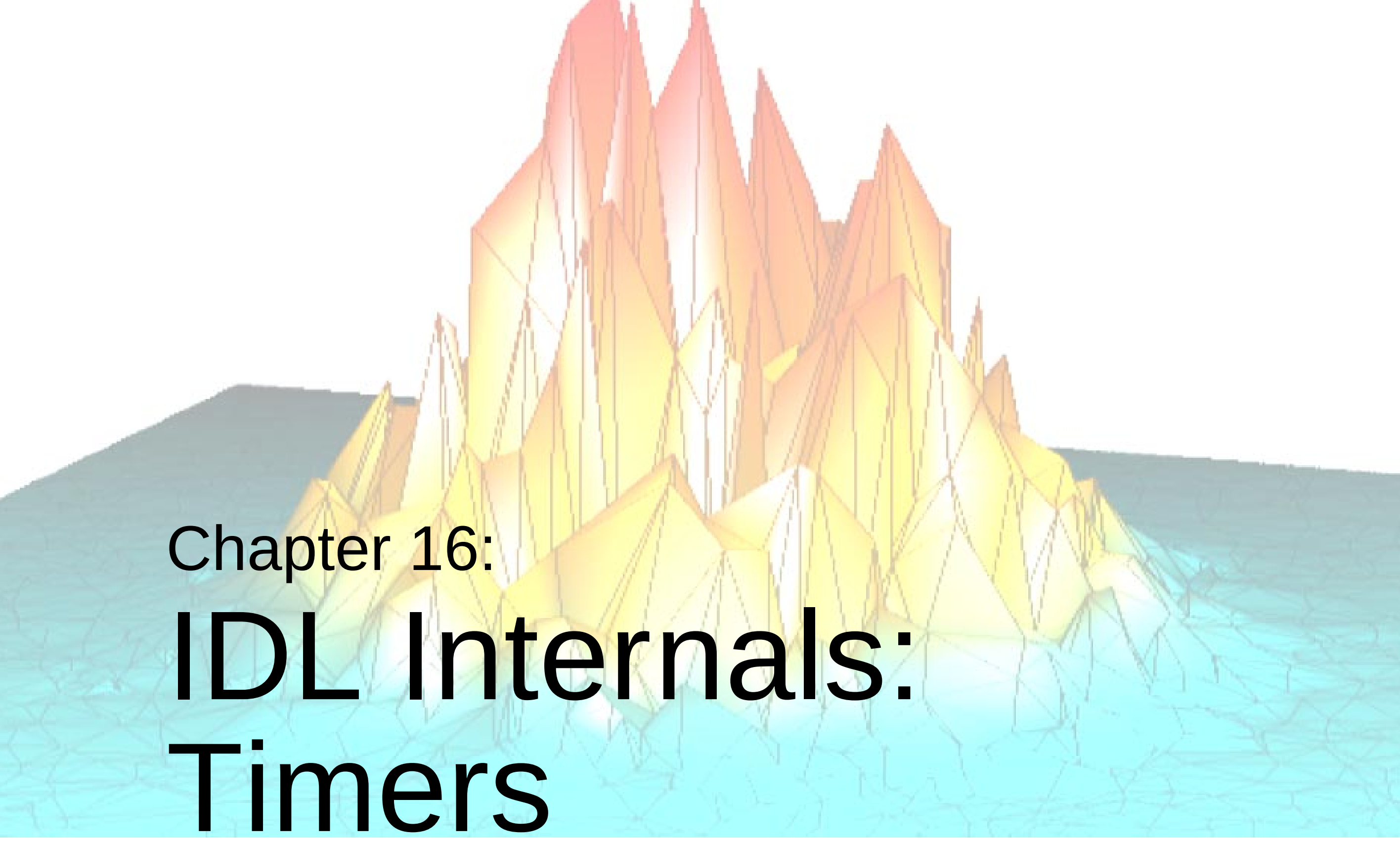
IDL_SignalSuspend() replaces the process signal mask with the ones in set and then suspends the process until a signal is delivered. On return, the original process signal mask is restored:

```
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 16: 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 constant 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 all users. 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 and VMS, 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 and Macintosh OS, 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 and VMS, 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.

Under Windows and the Macintosh OS, **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 and OpenVMS, **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.

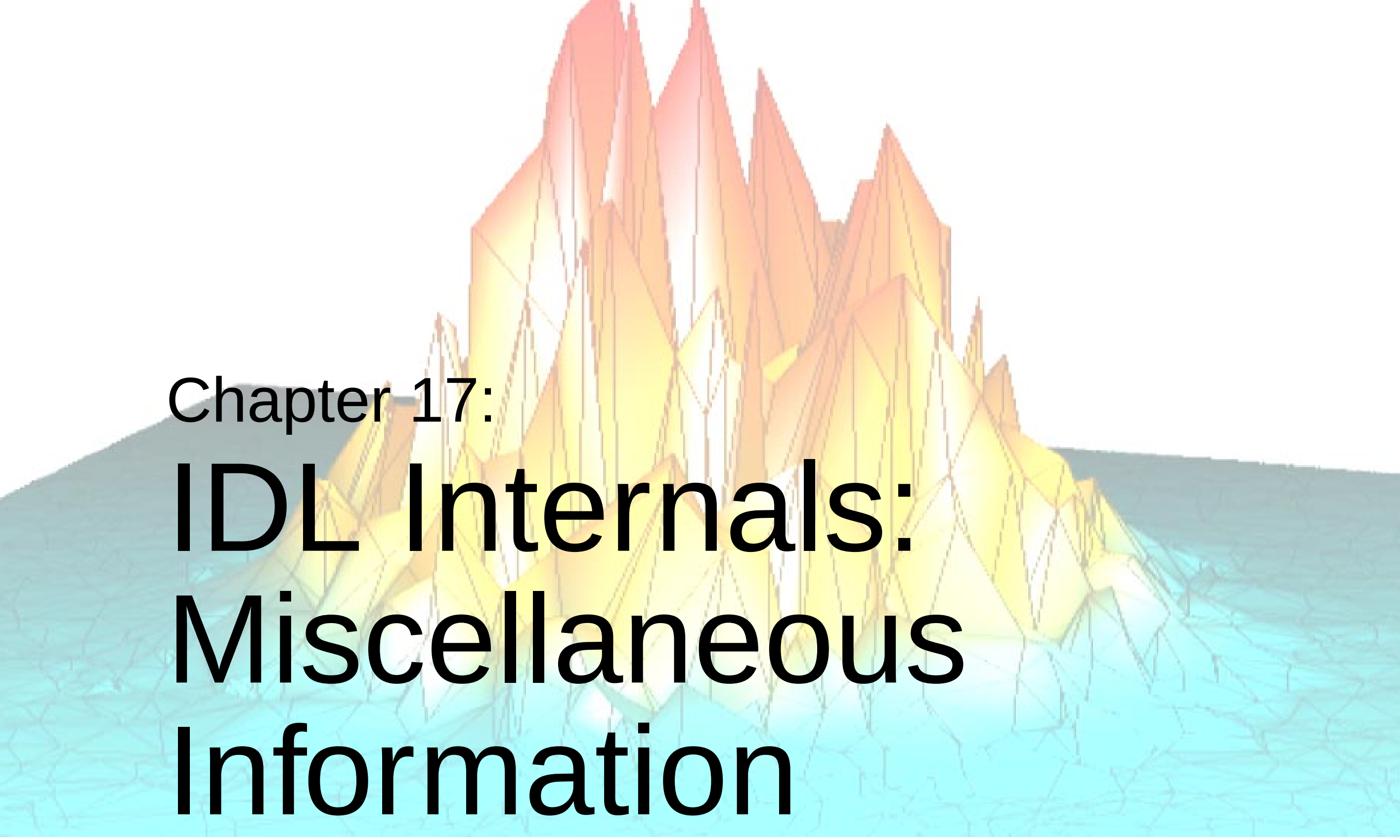
It has been our experience that some UNIX platforms have more problem with this issue than others. You should let experience guide you in deciding when to block signals and when to let them go. Input/Output to device special files under HP-UX and SGI IRIX are known to be especially vulnerable.

```
void IDL_TimerBlock(stop)
```

where:

stop

TRUE if the timer should be suspended, FALSE to restart it.



Chapter 17: 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.

Please note that system routines (routines added to IDL using LINKIMAGE or CALL_EXTERNAL) should not use the IDL dynamic memory routines. Instead, use **IDL_GetScratch()** (see "Getting Dynamic Memory" on page 188) which prevents memory from being lost under error conditions.

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(M_CNTGETMEM, action|IDL_MSG_ATTR_SYS, 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(M_CNTFREMEM, action|IDL_MSG_ATTR_SYS, 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_PRO_PTR proc)
```

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.

System Variables

The values of certain system variables are available globally to user programs. In all cases, these variables should be considered READ-ONLY. These variables are summarized in the following table.

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 17-1: IDL System Variables Available to User Programs

Functions for Returning System Variable Values

The following functions return the same information as the global system variables described above, but in a functional form. Note these values should be considered READ-ONLY.

IDL_STRING *IDL_SysvVersionArch(void)

This function returns a pointer to **IDL_SysvVersion.arch**.

IDL_STRING *IDL_SysvVersionOS(void)

This function returns a pointer to **IDL_SysvVersion.os.**

IDL_STRING *IDL_SysvVersionOSFamily(void)

This function returns a pointer to **IDL_SysvVersion.os_family.**

IDL_STRING *IDL_SysvVersionRelease(void)

This function returns a pointer to **IDL_SysvVersion.release.**

IDL_STRING *IDL_SysvDirFunc(void)

This function returns a pointer to **IDL_SysvDir.**

IDL_STRING *IDL_SysvErrStringFunc(void)

This function returns a pointer to **IDL_SysvErrString.**

IDL_STRING *IDL_SysvSyserrStringFunc(void)

This function returns a pointer to **IDL_SysvSyserrString.**

IDL_LONG IDL_SysvErrCodeValue(void)

This function returns the value of !ERR.

IDL_LONG IDL_SysvErrorCodeValue(void)

This function returns the value of !ERROR.

IDL_LONG IDL_SysvOrderValue(void)

This function returns the value of !ORDER.

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 (the Macintosh OS and 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.

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.

char *IDL_FileTermName(void)

This function returns the value of **IDL_FileTerm.name**. This function is only available under UNIX and OpenVMS.

int IDL_FileTermIsTty(void)

This function returns the value of **IDL_FileTerm.is_tty**. This function is only available under UNIX and OpenVMS.

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 160). 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 host[64];             /* Machine name */
  char wd[IDL_MAX_PATH];     /* Working Directory */
char date[25];               /* Current System Time */
} IDL_USER_INFO;
```

Constants

Preprocessor constants defined in the `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, many 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.

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 `export.h` handle recurring small jobs. Those macros not discussed elsewhere in this book are covered here.

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 an expression.

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 do not call it with an 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_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)

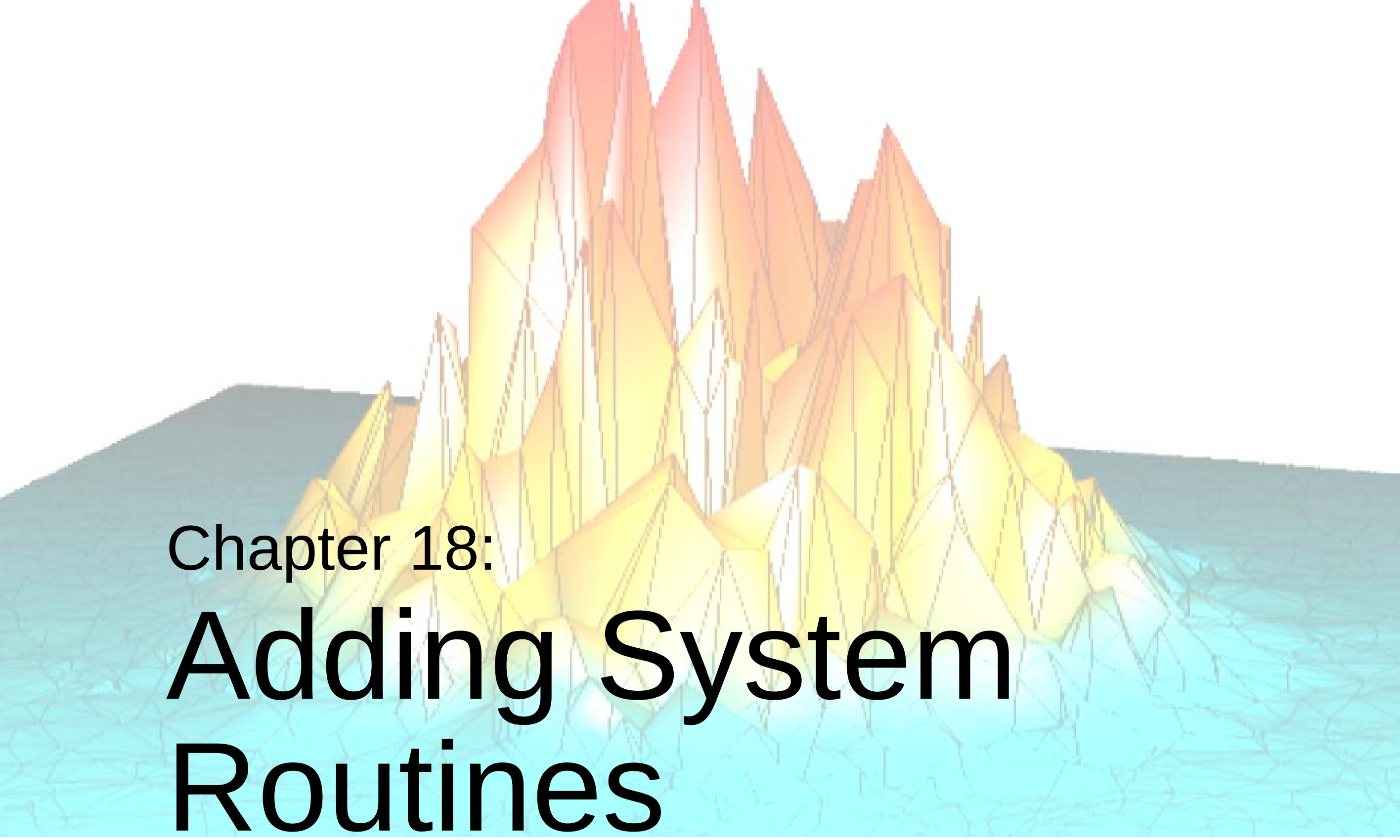
IDL_CHARA() takes the address of its argument and casts it to be a pointer to **char**. It is used to get a generic pointer to a memory location.

IDL Global Data Under VAX/OpenVMS

Under VAX/OpenVMS, IDL's global variables are available as linker UNIVERSAL symbols. However, the locations of these symbols within the IDL sharable image change from release to release. Therefore, if your program directly accesses these symbols, you must re-link your application every time you install a new IDL distribution.

However, it is possible to minimize the problem of re-linking with each IDL release by using the functions—described in "Functions for Returning System Variable Values" on page 284, "Functions for Returning IDL_FileTerm Variable Values" on page 286, and "Functions for Returning Data Type Variable Values" on page 288—that also provide access to global data. These functions are found in the IDL.EXE transfer vector. Therefore, if the functions are used, no re-linking is needed between releases as long as the transfer vector is not changed. Although Research Systems cannot always avoid changing the transfer vector, it is less likely to change than the locations of UNIVERSAL symbols.

Under ALPHA/OpenVMS, global variables are found in the SYMBOL_VECTOR just like the exported functions, so the previously-described VAX/OpenVMS problem does not occur with ALPHA/OpenVMS. Under ALPHA/OpenVMS, accessing the global variable is equivalent to using the function.



Chapter 18:

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 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).

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 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_KWGetParams()**, which processes the keywords in a standard way. For more information on keywords, see "IDL Internals: Keyword Processing" on page 197.

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 "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, say, **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.

C

```
 1 #include <stdio.h>
 2 #include "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)
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 }
```

Figure 18-1: 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 236 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 173 and "Temporary Variables" on page 181.

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.

4

`nr.h` is the header file provided with Numerical Recipes in C code.

7

FZROOTS2 has the usual three standard arguments.

C

```c
#include <stdio.h>
#include <stdarg.h>
#include "export.h"
#include <nr/nr.h>


IDL_VPTR fzroots2(int argc, IDL_VPTR *argv, char *argk)
{
  static int force_type;
  static IDL_LONG do_double;
  static double eps;
  static IDL_LONG no_polish;
  static IDL_VPTR tc_input;
  static IDL_KW_PAR kw_pars[] = {
    {"DOUBLE", IDL_TYP_LONG, 1, 0, &force_type,
     IDL_CHARA(do_double) },
    { "EPS", IDL_TYP_DOUBLE, 1, 0, 0, IDL_CHARA(eps) },
    { "NO_POLISH", IDL_TYP_LONG, 1, IDL_KW_ZERO, 0,
      IDL_CHARA(no_polish) },
    { "TC_INPUT", 0, 1, IDL_KW_OUT|IDL_KW_ZERO, 0,
      IDL_CHARA(tc_input) },
    { NULL }
  };

  IDL_VPTR result;
  IDL_VPTR c_raw;
  IDL_VPTR c_tc;
  IDL_MEMINT m;
  void *outdata;
  IDL_MEMINT dim[IDL_MAX_ARRAY_DIM];
  int rtype;
  static IDL_ALLTYPES zero;


  eps = 2.0e-6;
  (void) IDL_KWGetParams(argc, argv, argk, kw_pars,&c_raw,1);

  IDL_ENSURE_ARRAY(c_raw);
  IDL_ENSURE_SIMPLE(c_raw);
  if (c_raw->value.arr->n_dim != 1)
  IDL_Message(IDL_M_NAMED_GENERIC, IDL_MSG_LONGJMP,
              "Input argument must be a column vector.");
  m = c_raw->value.arr->dim[0];
  if (--m <= 0)
    IDL_Message(IDL_M_NAMED_GENERIC, IDL_MSG_LONGJMP,
                "Input array does not have enough elements");

  if (tc_input)
    IDL_StoreScalar(tc_input, IDL_TYP_LONG, &zero);


  if (force_type) {
    rtype = do_double ? IDL_TYP_DCOMPLEX : IDL_TYP_COMPLEX;
  } else {
    rtype = ((c_raw->type == IDL_TYP_DOUBLE)
             || (c_raw->type == IDL_TYP_DCOMPLEX))
      ? IDL_TYP_DCOMPLEX : IDL_TYP_COMPLEX;
  }
```

Figure 18-2: fzroots2.c

```
     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) {
C    70     zroots_f((fcomplex *) c_tc->value.arr->data, m,
     71              ((fcomplex *)outdata)-1,!no_polish,(float)eps);
     72   } else {
     73     zroots_d((dcomplex *) c_tc->value.arr->data, m,
     74              ((dcomplex *) outdata) - 1, !no_polish, eps);
     75   }
     76
     77   if (tc_input) IDL_VarCopy(c_tc, tc_input);
     78   else if (c_raw != c_tc) IDL_Deltmp(c_tc);
     79
     80   return result;
     81 }
```

Figure 18-2: fzroots2.c

9

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.

11

The value of the EPS keyword.

12

The value of the NO_POLISH keyword.

13

The value of the TC_INPUT keyword.

14

This array defines the keywords accepted by FZ_ROOTS2.

15

Since setting DOUBLE to 0 has a different meaning than not specifying the keyword at all, **force_type** is used to detect the fact that the keyword is set independent of its value.

17

The EPS keyword allows the user to specify the **eps** tolerance parameter. **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 **eps**, otherwise **eps** will not be changed.

18

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.

20

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.

25

This variable will receive the function result.

26

The input argument prior to any type conversion.

27

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.

28

The value of *m* to be passed to **zroots()**.

29

Pointer to the data area of the result variable. We declare it as `(void *)` so that it can point to data of any type.

30

Used to specify dimensions of the result. This will always be a vector of *m* elements.

31

IDL type code for result variable.

32

Used by **IDL_StoreScalar()** to type check the TC_INPUT keyword. It is declared as static to ensure it is initialized to zero.

35

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 **eps** and will be passed to **zroots()** unaltered.

36

Perform keyword processing.

38 – 39

Ensure that the input argument is an array, and is one of the basic types (not a file variable or structure).

40– 42

The input variable must be a vector, and therefore should have only a single dimension.

43 – 46

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.

48

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, **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 61.

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

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 **result_var()** shown in the following figure. The result of RSUM always has the same general shape and dimensions as the input argument. **result_var()** 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.

```
 char *result_var(IDL_VPTR template, int type, IDL_VPTR *res)
 /*
 * Allocate a result variable, using the template IDL_VPTR to determine
 * the structure, and type to determine the type. *res is set to
 * the new variable, and a pointer to its data area is returned.
 */
{
 char *data;
 IDL_VPTR lres;

 if (template->flags & IDL_V_ARR) {
   data = IDL_MakeTempArray(type, template->value.arr->n_dim,
                            template->value.arr->dim, IDL_ARR_INI_NOP, res);
   } else {
     lres = *res = IDL_Gettmp();
     lres->type = type;
     data = (char *) &(lres->value.c);
   }

 return data;
}
```

Figure 18-3: result_var() function for RSUM example

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:

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.

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 *) result_var(v, IDL_TYP_FLOAT, &r);
18   f_src = (float *) v->value.arr->data;
19   n = v->value.arr->n_elts - 1;
20   *f_dst++ = *f_src++;/* First element */
21   for (; n--; f_dst++) *f_dst = *(f_dst - 1) + *f_src++;
22 
23   return r;
24 }
```

Figure 18-4: Code for IDL_rsum1()

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.

18

Get a pointer to the data area of the input variable. At this point we know this variable is always a floating point array.

19

The number of data elements in the input.

20–21

The running sum computation.

23

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 code for RSUM2 is given in the following figure.

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 *) result_var(v, IDL_TYP_FLOAT, &r);
21   }
22 
23   *f_dst++ = *f_src++;/* First element */
24   n--;
25   for (; n--; f_dst++) *f_dst = *(f_dst - 1) + *f_src++;
26 
27   return r;
28 }
```

Figure 18-5: 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_GetVarData() 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.

15–21

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:

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.

35-37

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.

```
     1 cx_public IDL_VPTR IDL_rsum3(int argc, IDL_VPTR argv[])
     2 {
     3    IDL_VPTR v, r;
     4    union {
     5      char *sc;                 /* Standard char */
     6      UCHAR *c;                 /* IDL_TYP_BYTE */
     7      short *i;                 /* IDL_TYP_INT */
     8      IDL_UINT *ui;             /* IDL_TYP_UINT */
     9      IDL_LONG *l;              /* IDL_TYP_LONG */
    10      IDL_ULONG *ul;            /* IDL_TYP_ULONG */
    11      IDL_LONG64 *l64;          /* IDL_TYP_LONG64 */
    12      IDL_ULONG64 *ul64;        /* IDL_TYP_ULONG64 */
    13      float *f;                 /* IDL_TYP_FLOAT */
    14      double *d;                /* IDL_TYP_DOUBLE */
C   15      IDL_COMPLEX *cmp;         /* IDL_TYP_COMPLEX */
    16      IDL_DCOMPLEX *dcmp;       /* IDL_TYP_DCOMPLEX */
    17 } src, dst;
    18 IDL_LONG n;
    19

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 */

/* Get a result var, reusing the input if possible */
if (v->flags & IDL_V_TEMP) {
  r = v;
  dst = src;
} else {
  dst.sc = result_var(v, v->type, &r);
}

#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

C  switch (v->type) {
   DOCASE(IDL_TYP_BYTE, c);
   DOCASE(IDL_TYP_INT, i);
   DOCASE(IDL_TYP_LONG, l);
   DOCASE(IDL_TYP_FLOAT, f);
   DOCASE(IDL_TYP_DOUBLE, d);
   DOCASE_CMP(IDL_TYP_COMPLEX, cmp);
   DOCASE_CMP(IDL_TYP_DCOMPLEX, dcmp);
   DOCASE(IDL_TYP_UINT, ui);
   DOCASE(IDL_TYP_ULONG, ul);
   DOCASE(IDL_TYP_LONG64, l64);
   DOCASE(IDL_TYP_ULONG64, ul64);
   default: IDL_Message(IDL_M_NAMED_GENERIC, IDL_MSG_LONGJMP,
                        "unexpected type");
   }
#undef DOCASE
#undef DOCASE_CMP

return r;
}
```

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 on page 331.

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.

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_FUN_RET)();

typedef struct {
  IDL_FUN_RET 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 169.

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 "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_FUN_RET) 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_FUN_RET 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 18-1: 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 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_FUN_RET 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 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.

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

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 318, 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) which implements it.

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.

Example

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.

- There are instances where unrelated routines both use a common third party library, but they require different versions of this library (e.g. 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 Research Systems.

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 thus defined to the table of known routines as “stubs”. Stubs are entries in the system routine dispatch table that 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. 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 333.

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 from the module are made immediately once it is loaded.

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 users 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.

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.

C

```c
#include <stdio.h>
#include “export.h”

/* Handy macro */
#define ARRLEN(arr) (sizeof(arr)/sizeof(arr[0]))

/* 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_FUN_RET) 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”, ARRLEN(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, ARRLEN(function_addr))
    && IDL_SysRtnAdd(procedure_addr, FALSE, ARRLEN(procedure_addr));
}
```

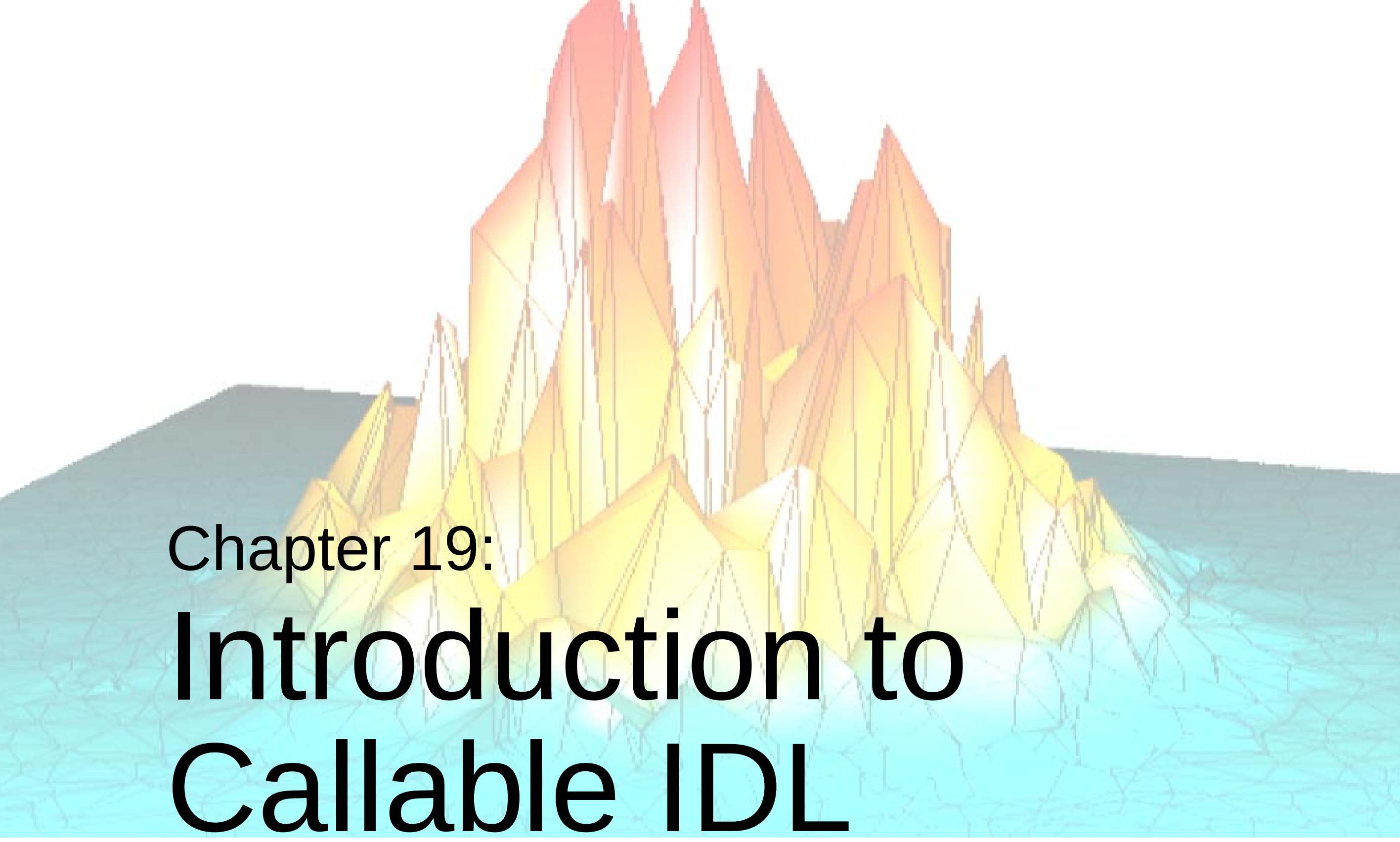
Figure 18-6: testmodule.c

- 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 19:

Introduction to Callable IDL

This chapter discusses the following topics:

Callable IDL

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 via CALL_EXTERNAL or LINKIMAGE.

This chapter provides a basic description of Callable IDL. Subsequent chapters discuss the specifics of using Callable IDL under UNIX and VMS ("Using Callable IDL Under UNIX and VMS" on page 349) and under Microsoft Windows ("Using Callable IDL Under Windows" on page 375). "AppleScript Support" on page 91, discusses "calling" IDL for Macintosh via AppleScript.

How Callable IDL is Implemented

IDL for Windows, IDL for UNIX, and IDL for VMS are packaged in a sharable form that allows other programs to call IDL as a subroutine. The details of packaging differ between platforms:

- IDL for Windows has a small driver program linked to a Dynamic Link Library (DLL).
- IDL for UNIX has a small driver program linked to a sharable object library.
- IDL for VMS is a sharable executable.

In all three cases, it is possible to link the sharable portion of IDL into your own programs. Note that Callable IDL is *not* a separate object that implements a library version of IDL. Interactive IDL as seen by the user calls the sharable IDL library itself.

IDL for Macintosh, is “callable” using AppleScript. See “AppleScript Support” on page 91.

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 "Inter-language Communication Techniques Which are Supported" on page 15 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 24.

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.

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: Signals" on page 259) 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 271.

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 “IDL Graphics Devices in Chapter 8 of the *IDL Reference Guide* 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 395, 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 3, “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 Research Systems, 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 Research Systems 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 was not designed to be used in a threaded program, nor is it threaded itself. Attempting to integrate IDL in a threaded application 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 Research Systems 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.

Initialization

Before calling IDL to execute instructions, you must initialize it using **IDL_Init()** (or **IDL_Win32Init()** for Windows applications). This is a one-time operation, and must occur before calling any other IDL function. Before writing your application, read carefully the platform-specific initialization details in the following chapters. (UNIX and VMS users see "Initialization" on page 351. Windows users see "Initialization" on page 377.)

Call IDL

Once Callable IDL is initialized, you can perform two types of operations:

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 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. (UNIX and VMS users see "Executing IDL Statements" on page 355. Windows users see "Executing IDL Statements" on page 380.)
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 186.)
   - Accessing data within IDL. (See "Looking Up Variables in Current Scope" on page 196.)

- Changing items in the process, such as signal handling or timers. (See "IDL Internals: Signals" on page 259, or "IDL Internals: Timers" on page 271.)
- Redirecting IDL output to your own function for processing. (UNIX and VMS users see "Diverting IDL Output" on page 353. Windows users see "Diverting IDL Output" on page 378.)

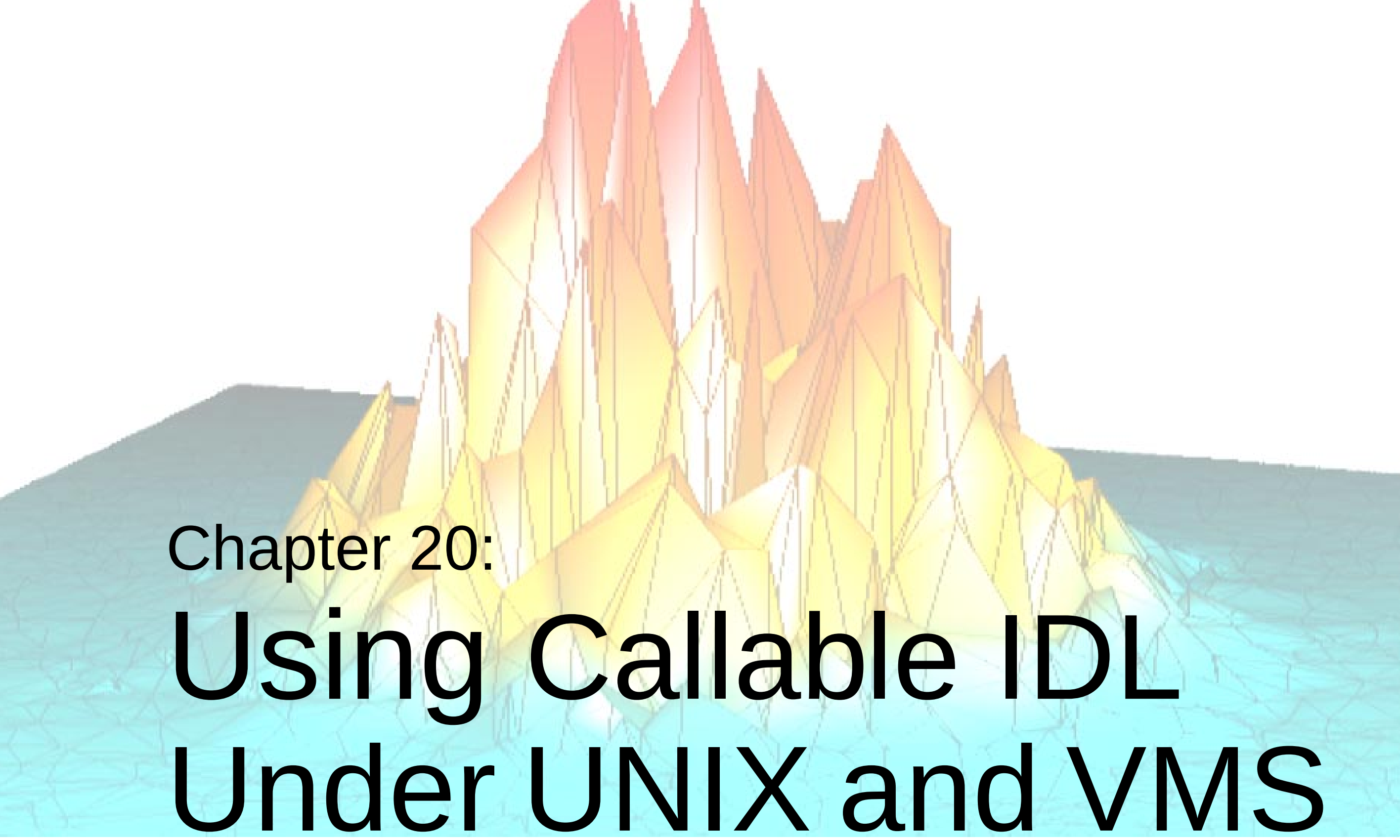
The above list is not complete, but is representative of the possibilities afforded by Callable IDL.

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. (UNIX and VMS users see "Cleanup" on page 358, Windows users see "Cleanup" on page 383.)

Documentation for Callable IDL

The following chapters discuss the specifics of using Callable IDL under UNIX and VMS ("Using Callable IDL Under UNIX and VMS" on page 349) and under Microsoft Windows ("Using Callable IDL Under Windows" on page 375). "AppleScript Support" on page 91 discusses "calling" IDL for Macintosh via AppleScript.



Chapter 20: Using Callable IDL Under UNIX and VMS

This chapter discusses the following topics:

Callable IDL and UNIX and VMS

This chapter discusses the procedures used when calling IDL's sharable libraries under UNIX and VMS. If you have not yet read "Introduction to Callable IDL" on page 339, please do so before continuing.

Procedures used when calling the IDL DLL under Microsoft Windows are covered in "Using Callable IDL Under Windows" on page 375. Procedures used when "calling" IDL for Macintosh using AppleScript are covered in "AppleScript Support" on page 91.

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 LINKIMAGE is not supported and is certain to corrupt and/or crash the IDL process.

Initialization

The **IDL_Init()** function prepares Callable IDL for use. This must be the first IDL routine called.

```
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 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 Windows.

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. Note that **IDL_Execute()** and **IDL_ExecuteStr()** are disabled when IDL is initialized with this option.

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.

Diverting IDL Output

When using a tty-based interface (UNIX or VMS), IDL sends its output to the screen for the user to see. When using a GUI based interface (Windows, Macintosh, or UNIX), the output goes to the 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.

Warning

Do not pop an output function you did not push. It is an error to attempt to remove the last remaining function.

```
IDL_TOUT_OUTF IDL_ToutPop(void);
```

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.

Warning

If the IDL_INIT_EMBEDDED option to **IDL_Init()** is set, **IDL_Execute()** and **IDL_ExecuteStr()** are disabled. This is also true when using an IDL student edition license.

IDL_Execute()

IDL_Execute() executes the command strings in the order given. It returns the value of !ERROR after the final command has executed. If the value of !ERROR 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 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 (using either the IDL_INIT_RUNTIME or IDL_INIT_EMBEDDED option) **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.

Warning

If either the IDL_INIT_EMBEDDED option or the IDL_INIT_RUNTIME option to **IDL_Init()** is set, **IDL_Execute()** and **IDL_ExecuteStr()** are disabled.

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.

Interactive IDL

IDL_Main() implements IDL as seen by the interactive user. In the interactive version of IDL as shipped by Research Systems, 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 arise if the different compilers use different naming conventions. For example, the SunOS Fortran compiler adds an underscore to the end of each name, while the C compiler does not. To call a Fortran routine from C, you must 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.

  Note that 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 Statements” on page 374 for examples showing 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 374 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 "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 }
```

Figure 20-1: Calling IDL from C

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:

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 !ERROR to zero so previous errors don’t confuse our results.

```
 1 #include <stdio.h>
 2 #include "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("!ERROR=0");
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 }
```

Figure 20-2: call_idl_fft()

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, 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 374 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:

1-27

In order to print variables returned from IDL, we must define a Fortran structure type for **IDL_VARIABLE**s. 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()**.

f77

```fortran
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.

     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)

```

Figure 20-3: Calling IDL from Fortran

f77

```
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

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)

```

Figure 20-3: Calling IDL from Fortran

f77

```
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))


C  Null Terminate command strings and store the address
C  for each command string in CMD_ARGV

        DO I = 1, 5
           WIDGET_CMDS(I)(MAXLEN:MAXLEN) = CHAR(0)
           CMD_ARGV(I) = LOC(WIDGET_CMDS(I))
        ENDDO

C  Execute a small widget program. Pressing the button allows
C  the program to end

        CALL IDL_Execute(%VAL(5), CMD_ARGV)

C  Shut down IDL
        CALL IDL_Cleanup(%VAL(0))

      ENDIF

   END
```

Figure 20-3: Calling IDL from Fortran

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.

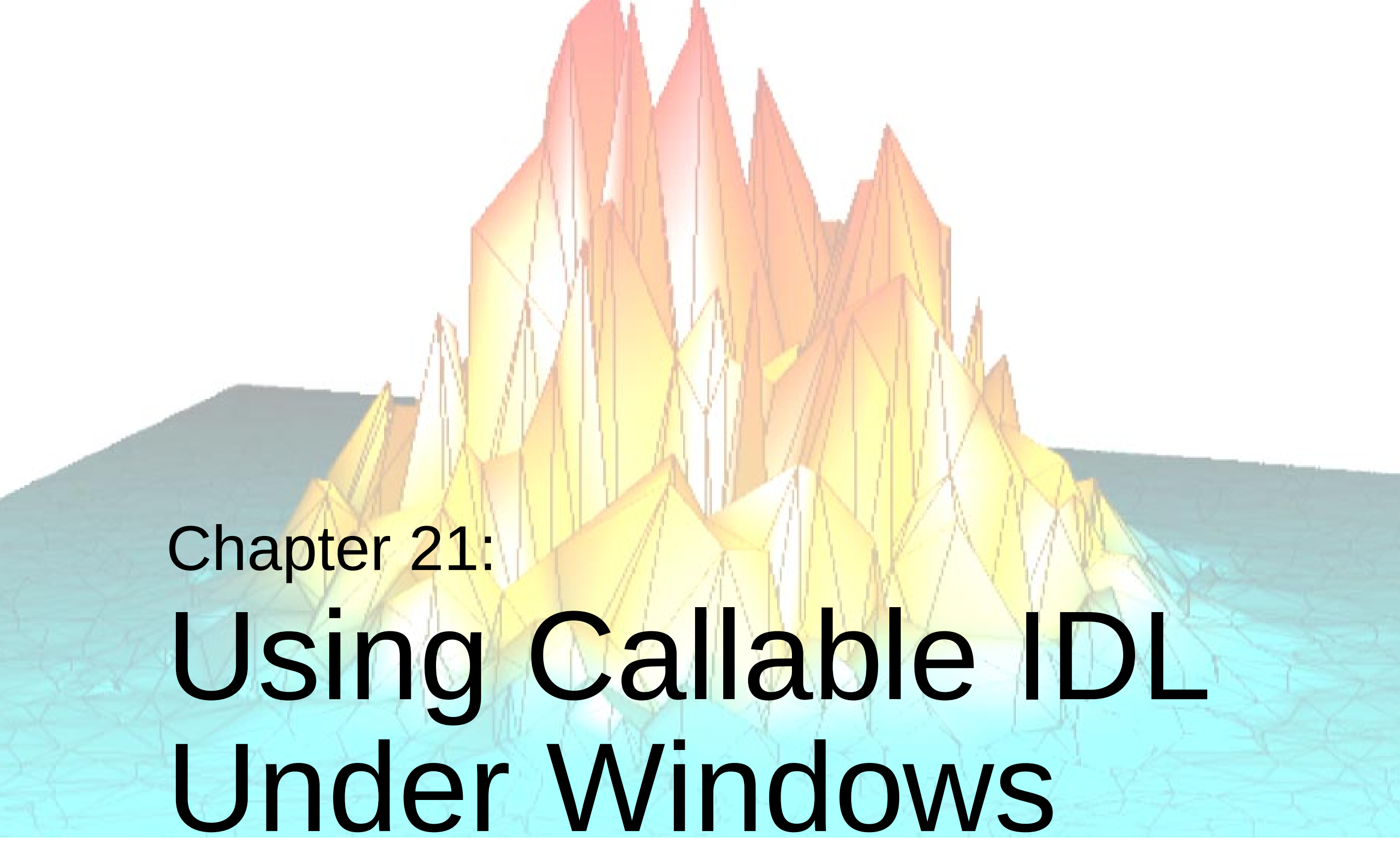
Compilation and Linking Statements

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 also contains a make rule for building the `calltest` application. The text of `calltest_unix.txt` is derived from those files.

Compilation and linking statements used when calling IDL on a VMS system are included in `make_vms.com`, a VMS command file located in the `callable` subdirectory of the `external` subdirectory of the main IDL directory.



Chapter 21: Using Callable IDL Under Windows

This chapter discusses the following topics:

Callable IDL and Windows

This chapter discusses calling the IDL Win32 DLL under Microsoft Windows. If you have not yet read "Introduction to Callable IDL" on page 339, please do so before continuing. The first sections of this chapter describe calling IDL from a 32-bit Win32 Windows application. Note that IDL does not support calls from 16-bit Windows applications, or from applications built using 32-bit APIs other than Win32.

Procedures used when calling the IDL for UNIX and IDL for VMS sharable objects are covered in "Using Callable IDL Under UNIX and VMS" on page 349. Procedures used when "calling" IDL for Macintosh using AppleScript are covered in "AppleScript Support" on page 91.

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 LINKIMAGE is not supported and is certain to corrupt and/or crash the IDL process.

Initialization

The **IDL_Win32Init()** function prepares the IDL DLL for use. **IDL_Win32Init()** must be called before any other function except **IDL_ToutPush()**.

```
int IDL_Win32Init(int iOpts, void *hinstExe, void *hwndExe,
                  void *hAccel);
```

where:

iOpts

Reserved. This argument should always be 0 (zero).

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 (UNIX or VMS), 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 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. **Caution:** Do not pop an output function you did not push. It is an error to attempt to remove the last remaining function.

```
IDL_TOUT_OUTF IDL_ToutPop(void);
```

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.

Warning

If IDL is called with either a runtime or embedded license, **IDL_Execute()** and **IDL_ExecuteStr()** are disabled.

IDL_Execute()

IDL_Execute() executes the command strings in the order given. It returns the value of !ERROR after the final command has executed. If the value of !ERROR 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 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, 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.

Warning

If IDL is called with a student license, **IDL_Execute()** and **IDL_ExecuteStr()** are disabled.

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.

This call should be placed in your Main **WndProc** and be called as a result of the **WM_CLOSE** message.

```
switch(msg){
           ...
      case WM_CLOSE:
           IDL_Cleanup(TRUE);
           any additional processing
           ...
```

Building an Application that Calls IDL

To build your 32-bit, Win32 application that calls IDL, you must take the following steps:

1. Include `exports.h`, found in the `external` subdirectory of the IDL distribution, in your source code.
2. Compile your application.
3. Link your application with `IDL32.LIB`.
4. Place `IDL32.DLL` in a directory with your application or see the readme.txt file located in the *`RSI-directory/`*`external/callable` for an alternative method.

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.

```
1  /*-----------------------------------------------------------------
--
2  * simple.c Source code for sample IDL callable application
3  *
4  * Copyright (c) 1992-1995, Research Systems Inc.
9  *-----------------------------------------------------------------
*/
10 #include <windows.h>
11 #include <windowsx.h>
12 #include <ctl3d.h>
13 #include <string.h>
14 #include <stdio.h>
15 #include "simple.h"
16 #include "export.h"
17
18 /*-----------------------------------------------------------------
19  * WinMain
20  *
21  * This is the required entry point for all windows
applications.
22  *
23  * 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;
```

```
267
268    /* Turn off the READONLY bit and postpone redraw */
269    lRet = SendMessage(hwndLog, EM_SETREADONLY, FALSE, 0L);
270    lRet = SendMessage(hwndLog, WM_SETREDRAW, FALSE, 0L);
271
272   /* Get the length of the text in the log window*/
273   lBufflen = SendMessage (hwndLog, WM_GETTEXTLENGTH, 0, 0L);
274   lNumLines = SendMessage (hwndLog, EM_GETLINECOUNT, 0, 0L);
275   lFirstView = SendMessage (hwndLog, EM_GETFIRSTVISIBLELINE, 0, 0L);
276   lRet = SendMessage (hwndLog, EM_SETSEL, lBufflen, lBufflen);
277
278    /* If we are adding text, wParam will be 0 */
279  if(!wParam)
280     lRet = SendMessage (hwndLog, EM_REPLACESEL, 0, lParam);
281  else{
282     if (lNumLines > (lFirstView + nDisplayable)){
283          int        iLineLen = 0;
284          int        iChar;
285          int        iLines = 0;
286          lNumLines--;
287          while(!iLineLen){
288              iChar = SendMessage(hwndLog, EM_LINEINDEX,
289                      (WPARAM)lNumLines, 0L);
290              iLineLen = SendMessage(hwndLog, EM_LINELENGTH,
291                      iChar, 0L);
292              if(!iLineLen)
293                  lNumLines--;
294          }
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

`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

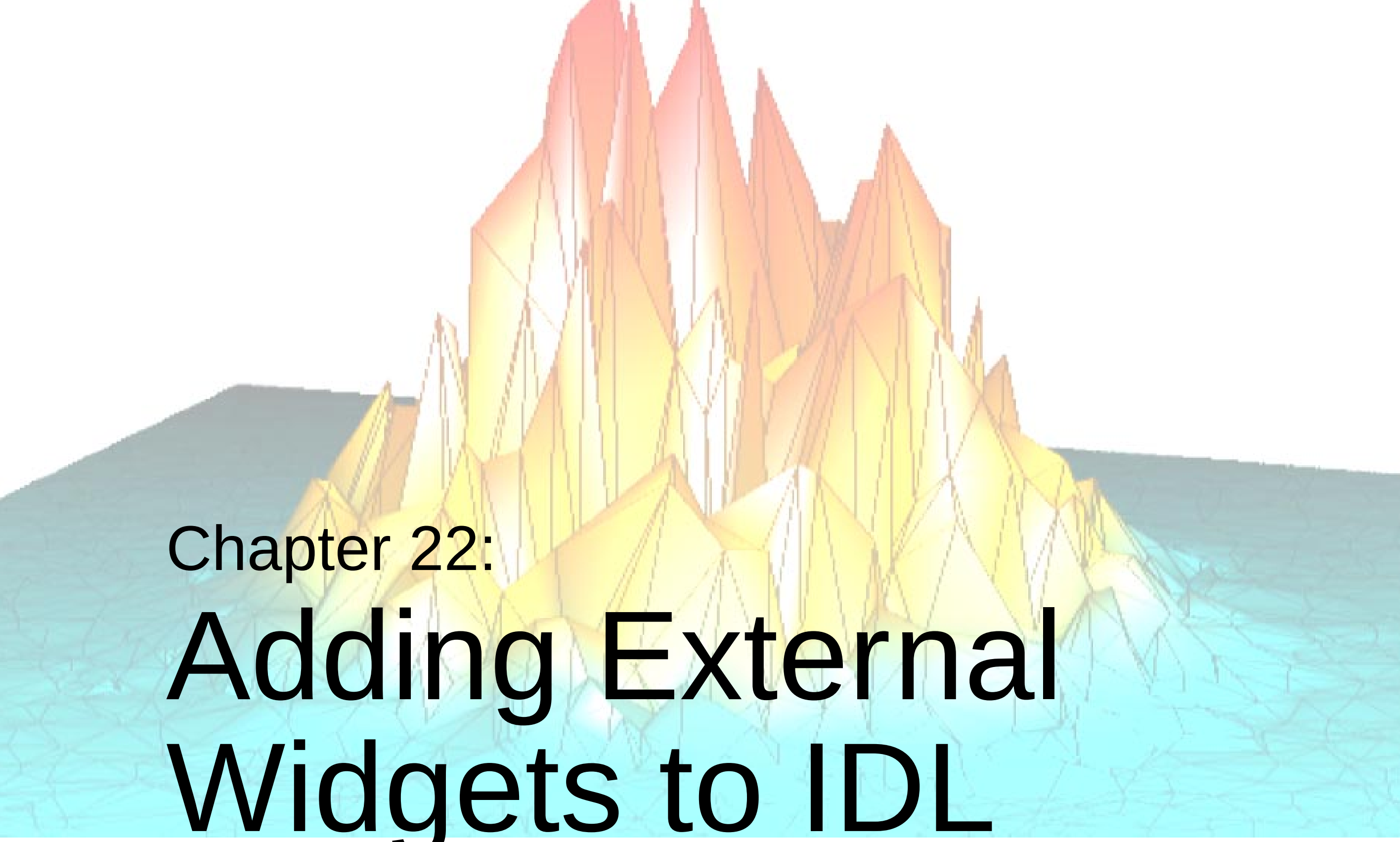
OutputMessage handles displaying the text to the output window. Since this window is a multi-line edit control, we have created it as a read-only window. See the source code for additional information on handling this situation.

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 22:

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, and Callable IDL users to add their own widgets to IDL widget hierarchies.

This feature does not always work with versions of IDL that statically link against the window system libraries, particularly those for Sun workstations. When Solaris 2.x ships from Sun with sharable Motif libraries, this limitation will disappear.

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 400 describes support functions that can be called from your external code to manipulate stub widgets. "Internal Callback Functions" on page 402 describes how to make stub widgets generate IDL widget events. Finally, "OpenVMS With WIDGET_STUB" on page 404 illustrates the use of stub widgets with an external program.

Note

IDL's WIDGET_STUB functionality was designed for the X/Motif windowing system, and is not supported under Microsoft Windows or on the Macintosh.

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 400.

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 but hides the bulk of the actual internals. These functions are exported by the IDL program for use by CALL_EXTERNAL code, but are not advertised in `export.h`.

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);
```

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.

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.

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.

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.

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.

OpenVMS With WIDGET_STUB

The following example adds the Motif ArrowButton widget to the OpenVMS version of IDL in the form of an IDL program named `widget_arrowb.pro`. It would be straightforward to do the same with any version of IDL supporting CALL_EXTERNAL.

The WIDGET_ARROWB widget implemented below 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.

The IDL Program for WIDGET_ARROWB

The following text is the IDL program for WIDGET_ARROWB. It should be saved in a file named `WIDGET_ARROWB.PRO`:

```
FUNCTION widget_arrowb, parent, use_own_size, $
   UVALUE=uvalue, _EXTRA=extra
parent = LONG(parent)
result = WIDGET_STUB(parent,_extra=extra)
if (N_ELEMENTS(uvalue) NE 0) THEN $
   WIDGET_CONTROL, result, SET_UVALUE=uvalue
JUNK = CALL_EXTERNAL('widget_arrowb','widget_arrowb', $
   def='diska:[idl.ali.arrowb].exe', parent, result, $
   use_own_size, value=[1, 1, 1])
RETURN, result
END
```

The C Program for widget_arrowb.c

The code invoked by the call to CALL_EXTERNAL 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:

```
/*
*
*
* arrowb.c - This file contains C code to be called from VMS 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.
```

```
*
*/
#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_dir:[external]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(int stub, int width, int height)
{
  char *stub_rec;
  unsigned long t_id, b_id;
  IDL_WidgetStubLock(TRUE);
  if (stub_rec = IDL_WidgetStubLookup(stub)) {
    IDL_WidgetGetStubIds(stub_rec, &t_id, &b_id);
    printf("Setting WIDGET %d to width %d and height %d\n",
           stub, width,height);
    XtVaSetValues((Widget) b_id, XmNwidth, width, XmNheight,
                  height, NULL);
  }
  IDL_WidgetStubLock(FALSE);
}
int widget_arrowb(IDL_LONG parent, IDL_LONG stub,
                  int 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;
}
```

Compiling and Linking the C File

This C file is compiled and linked into a sharable image usable by CALL_EXTERNAL by a DCL command file named `WIDGET_ARROWB.COM`:

```
$ if "’’f$search("SYS$SYSTEM:VAXVMSSYS.PAR")’" .eqs. ""
$ then
$ !     ALPHA
$       cc widget_arrowb.c
$       link/share widget_arrowb, sys$input/opt
/exe=widget_arrowb.exe
                IDL_DIR:[BIN.BIN_ALPHA]idl/share
                SYS$SHARE:DECW$XMLIBSHR12.EXE/SHARE
                SYS$SHARE:DECW$XTLIBSHRR5.EXE/SHARE
                SYS$SHARE:DECW$XLIBSHR/SHARE
                SYMBOL_VECTOR=(widget_arrowb=PROCEDURE)
$ else
$ !     VAX
$       cc widget_arrowb.c
$       link /share widget_arrowb, sys$input/opt
/exe=widget_arrowb.exe
        IDL_DIR:[BIN.BIN_VAX]IDL/SHARE
        SYS$SHARE:DECW$XMLIBSHR12.EXE/SHARE
        SYS$SHARE:DECW$XTLIBSHRR5.EXE/SHARE
        SYS$SHARE:DECW$XLIBSHR/SHARE
        sys$share:vaxcrtl/share
        universal = widget_arrowb
$ endif
```

Execute the file by entering:

```
$ @WIDGET_ARROWB
```

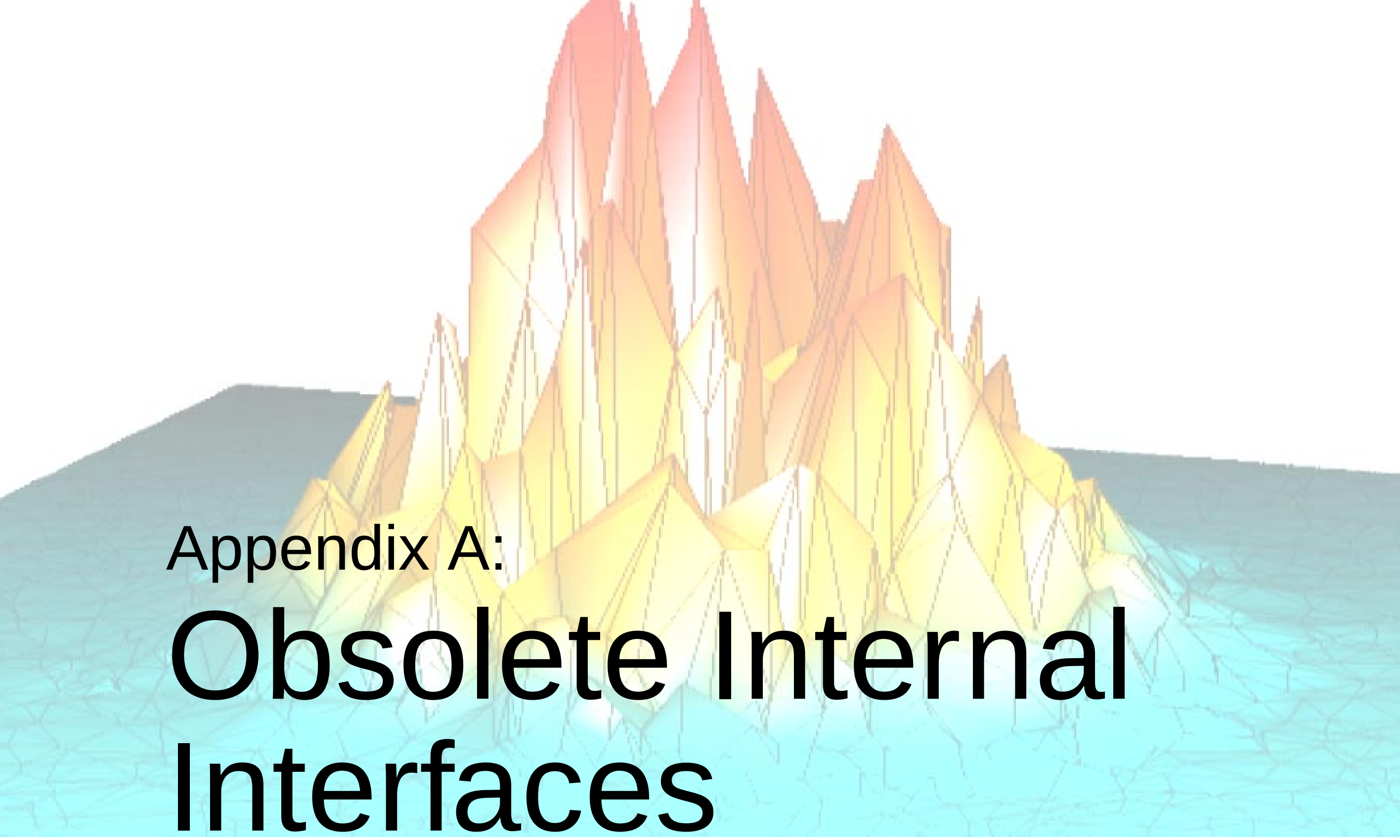
An IDL Program to Test the External Widget

Shown below is an IDL widget program to test the ARROWB widget. This program should be saved in a file called `TEST.PRO`. Note that one arrow button uses IDL's default sizing, while the other uses the WIDGET_ARROWB version:

```
PRO 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 test
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)
arrow_w = WIDGET_ARROWB(a, 1, XSIZE=100, YSIZE=50, UVALUE=label)
WIDGET_CONTROL, /REALIZE, a
XMANAGER, 'TEST', a
END
```

Start IDL and run the test program by entering:

```
$ IDL
IDL> TEST
```



Appendix A: Obsolete Internal Interfaces

This chapter discusses the following topics:

Interfaces Obsoleted in IDL 5.3

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 Research Systems. This section is supplied to help those maintaining older code. New code should be written using the information found in “Registering Routines” in Chapter 18.

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 169.

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 Research Systems. This section is supplied to help those maintaining older code. New code should be written using the information found in Chapter 18:, “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 414.)

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 197) 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:

$$\mathrm{BitMask} = 2^{\mathrm{TypeCode}}$$

or use the **IDL_TYP_MASK** macro (see “Type Masks” on page 161).

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 159.

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.

Compatibility with Versions 2 and 3

IDL for UNIX and IDL for VMS provide support for code written against the IDL internal interfaces of previous versions of IDL. IDL for Windows and IDL for Macintosh are new enough that there is no legacy code of this nature in existence, so they do not provide these interfaces.

Support for interface routines used by IDL Version 2 and IDL Version 3 is provided by a set of compatibility wrapper routines that recognize code written for the old interfaces and translates it into a form recognizable by the current interface. To use these compatibility routines, you should include the C header file `obsolete.h` at the top of your file:

```
#include <stdio.h>
#include "export.h"
#include "obsolete.h"
```

The `obsolete.h` file uses the C preprocessor to convert old constants, data types, and functions with variable numbers of arguments to their new names. It also contains prototypes for the compatibility wrapper functions.

Under UNIX, the compatibility wrapper functions are contained in an additional sharable library named `libidl_obsolete` that must be linked against your code. Under VMS, they are contained in the IDL sharable executable along with the current interface.

IDL Version 1 Compatibility

The routines described in this section provide compatibility with the interface routines used by IDL for VMS Version 1. These routines should not be used when writing new code. They are documented here for the sole purpose of making the port of older code to the current version easier.

Data Type Codes

IDL Version 1 used a Fortran include file named `TYPEDEF.PAR` to define the type codes as Parameter Statements. Here are the contents of a version of that file that can be used to port existing code to the current version of IDL:

```
c TYPEDEF.PAR (Fortran Include File)
c
c Define type codes for VMS IDL Version 2. Define both the
c old Version 1 names and the current Version 2 names.
c
c New type names:
Parameter TYP_BYTE = 1, TYP_INT = 2, TYP_LONG = 3
Parameter TYP_FLOAT = 4, TYP_DOUBLE = 5, TYP_COMPLEX = 6
Parameter TYP_STRING = 7, TYP_STRUCT = 8
c
c Version 1 type names with Version 2 values:
Parameter TBYTE = 1, TINT = 2, TLONG = 3
Parameter TFLT=4, TDOUBL=5, TCOMPLEX = 6, TASCII = 7
```

Both the old and the new data type names are defined. The actual values that define the type codes, as well as the variable structure, have been changed. Programs that symbolically refer to the type codes need only be recompiled. Note that the compatibility routines referred to in the previous section require the newer type code values.

Routines

The following table is a summary of the IDL Version 1 VMS interface routines documented in the file `WRITEYOUR.RNO` (which was distributed with IDL Version 1) and their implementation in newer versions:

Version 1 Routine	Changes
BYTE	Renamed BYTE_V1

Table 22-1: IDL for VMS Version 1 Routines Mapped to Later Versions of IDL.

Version 1 Routine	Changes
FIX	Renamed FIX_V1
FLOAT_IDL	
LONG	
DOUBLE	
COMPLEX	Renamed COMPLEX_V1
STRING	Renamed STRING_V1
CONVERT_TYPE	
FOR_CHKPAR	Routine must be called using CALL_V1
GETLON_SCL	
GETFLOAT_SCL	
GETDOUBLE_SCL	
FOR_GETSTRING	
FOR_GETTMP	
FOR_GETDIMS	
CREARR	
TSTDELTMP	
FOR_STORE_SCALAR	
COPYVAR	
PRINTERRMSG	
IDLERR	
IO_ERROR	
FOR_IO_ERROR	Not implemented
BYTE1	
INTEGER2	
INTEGER4	

Table 22-1: IDL for VMS Version 1 Routines Mapped to Later Versions of IDL.

Version 1 Routine	Changes
GET_KBINT	
STORE1	
STORE2	
STORE4	

Table 22-1: IDL for VMS Version 1 Routines Mapped to Later Versions of IDL.

Calling Convention

IDL Version 1 modules are defined as follows:

```
SUBROUTINE XXX(P1, ...., Pn)
```

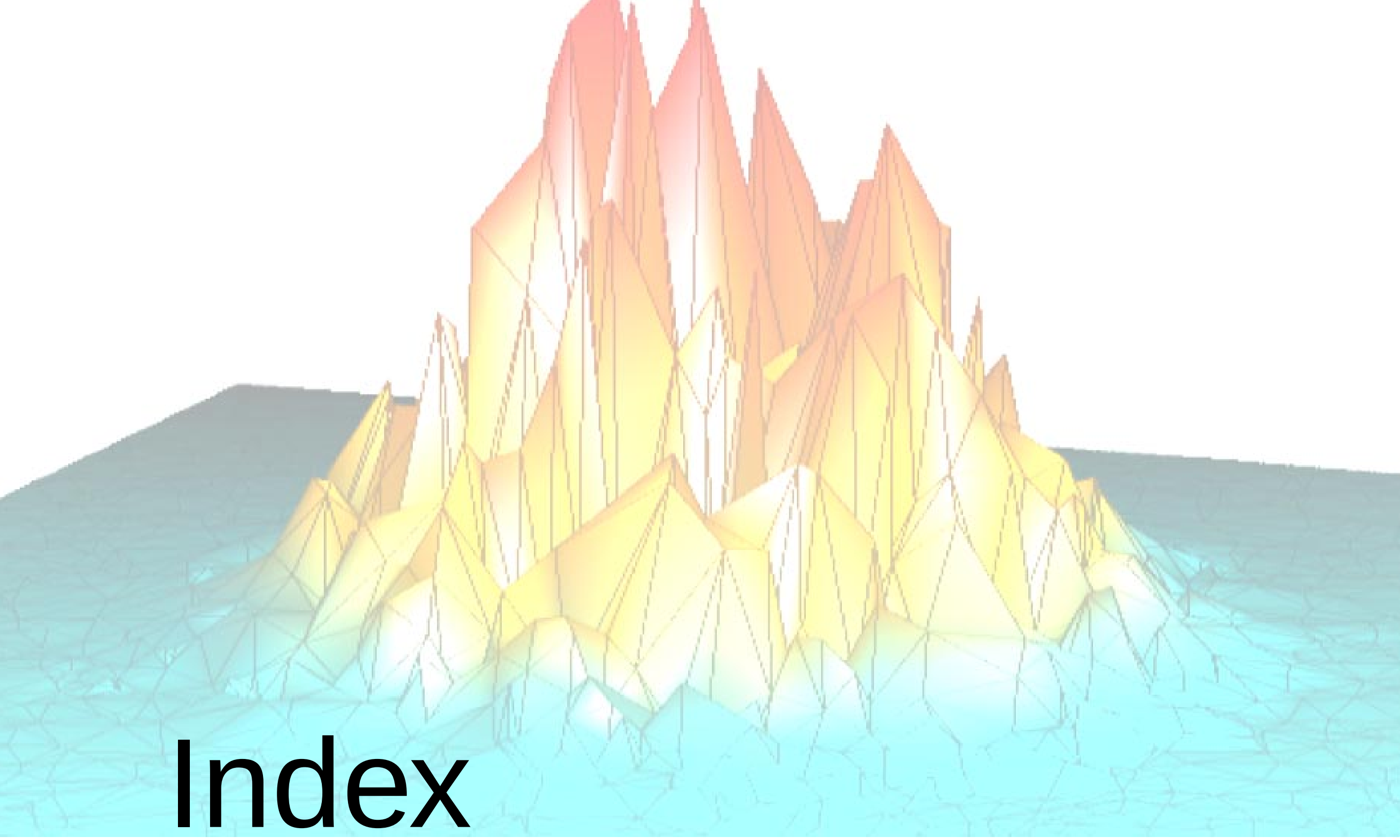
IDL Version 2 and later modules have the following Fortran definition:

```
SUBROUTINE XXX(N N, ARG LIST)
C ARG LIST is an array of pointers to VARIABLE structures:
INTEGER * 4 ARG LIST(*)
NARGS = %LOC(N N)
```

The routine CALL_V1 is provided in IDL Version 2 and later to convert the new calling sequence to the old. To use CALL_V1, you must write a short wrapper routine similar to the following, which calls the function XXX:

```
INTEGER*4 FUNCTION XXX WRAPPER( ARGC, ARGV )
EXTERNAL XXX
INTEGER*4 CALL_V1
XXX WRAPPER = CALL_V1(ARGC, ARGV, XXX)
RETURN
END
SUBROUTINE XXX(P1, P2, ..., Pn)
... ... ... Body of original routine XXX
```

The same method can be used to call procedures, which are defined as Fortran subroutines, rather than as functions.



Index