DECrpc

Programming Guide

Order Number: AA-PC2AA-TE
September 1990

Product Version:	DECrpc, Version 1.0 and higher
Operating System and Version:	VMS, Version 5.3 and higher

This manual provides information for programmers developing distributed applications based on DECrpc.

digital equipment corporation
maynard, massachusetts

Contents

About This Manual

1 Concepts and Terminology

3 Steps in Building a Distributed Application

4 Writing Interface Definitions

6 NIDL C Syntax

7 Special Topics

8 error_$ Routine Reference Pages

9 lb_$ Routine Reference Pages

10 pfm_$ Routine Reference Pages

11 pgm_$ Routine Reference Pages

12 rpc_$ Routine Reference Pages

13 rrpc_$ Routine Reference Pages

14 socket_$ Routine Reference Pages

15 uuid_$ Routine Reference Pages

16 Process and Utility Reference Pages

Glossary

Examples

Figures

Tables

About This Manual

This manual provides programming information for the Digital Remote Procedure Call (DECrpc) Version 1.0. The software is based on and is compatible with Version 1.5 of Apollo's Network Computing System (NCS).

This is a new manual in the DECrpc documentation set; the manual is based on the manual *NCS Reference* by Apollo Systems Division of Hewlett Packard.

Audience

This reference manual is for programmers developing applications based on DECrpc. If you are running, rather than developing, distributed applications, you should not need this book. *Guide to the Location Broker* explains how to establish and maintain the run-time support necessary for distributed applications.

In general, the body of this book shows examples written in C for the VMS operating system.

Organization

This manual contains sixteen chapters, a glossary, and an index.

Chapter 1 introduces DECrpc and the concept of a distributed application.

Chapter 2 surveys DECrpc software.

Chapter 3 introduces the steps in building a distributed application.

Chapter 4 describes how to define interfaces in Network Interface Definition Language (NIDL).

Chapter 5 describes how to develop distributed applications that use DECrpc.

Chapter 6 describes the C syntax of NIDL.

Chapter 7 describes special programming topics.

Chapter 8 includes a reference page for each `error_$` routine.

Chapter 9 includes a reference page for each `lb_$` routine.

Chapter 10 includes a reference page for each `pfm_$` routine.

Chapter 11 includes a reference page for the `pgm_$exit` routine.

Chapter 12 includes a reference page for each `rpc_$` routine.

Chapter 13 includes a reference page for each `rrpc_$` routine.

Chapter 14 includes a reference page for each `socket_$` routine.

Chapter 15 includes a reference page for each `uuid_$` routine.

Chapter 16 includes a reference page for each process and utility.

To submit comments on this document, please use the Reader's Comments form at the back of the book.

Related Documentation

For more information on topics related to NCS, see the following document:

Guide to the Location Broker

This book explains how to set up and administer the DECrpc run-time software, the Location Broker.

Conventions

The following conventions are used in this guide:

`special`	In text, each mention of a specific command, option, partition, pathname, directory, or file is presented in this type.
variable	In syntax descriptions, this type indicates terms that are variable.
`literal`	In syntax descriptions, this type indicates terms that are constant and must be typed just as they are presented.
[ ]	In syntax descriptions, brackets indicate terms that are optional.
. . .	In syntax descriptions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
UPPERCASE	The ULTRIX system differentiates between lowercase and uppercase characters. On ULTRIX systems, enter uppercase characters only where specifically indicated by an example or a syntax line.
`example`	In examples, computer output text is printed in this type.
`example`	In examples, user input is printed in this bold type.

new term	In text, new terms are introduced in this bold type.
$	This is the default user prompt in multiuser mode.
#	This is the default superuser prompt.
. . .	In examples, a vertical ellipsis indicates that not all of the lines of the example are shown.

Concepts and Terminology 1

This chapter describes the concepts and terminology of DECrpc, the remote procedure call mechanism supported by the VMS and ULTRIX operating systems. DECrpc is based on and is compatible with the RPC component of Apollo's Network Computing System (NCS) Version 1.5. NCS is a set of tools for heterogeneous distributed computing.

1.1 Distributed Applications

Using remote procedure calls, software applications can be distributed across heterogeneous collections of computers, networks, and programming environments. Distributed applications can take advantage of computing resources throughout a network or internet, with different parts of each program executing on the computers best suited for the tasks.

There are many applications that can be distributed among multiple systems. For example, one program might perform graphical input and output on a workstation while it does intense computation on a supercomputer. A program that performs many independent calculations on a large set of data could distribute these calculations among any number of available processors on the network or internet.

1.2 RPC Software

The software for writing distributed applications is written in portable C wherever possible. The components are:

- Remote Procedure Call (RPC) run-time library
- Network Interface Definition Language (NIDL) Compiler
- Location Brokers

The RPC run-time library and the Location Brokers provide run-time support for distributed applications.

The NIDL Compiler is a tool for developing distributed applications.

1.2.1 RPC Run-time Library

The DECrpc run-time library provides the routines that enable local programs to execute procedures on remote hosts. These routines transfer requests and responses between the programs calling the procedures and the programs executing the procedures.

When you develop distributed applications, you usually do not use many run-time routines directly. Instead, you write interface definitions in Network Interface Definition Language and use the NIDL Compiler to generate most of the required calls to the run-time library.

1.2.2 Location Broker

A **broker** is a server that provides information about resources. The Location Broker enables clients to locate specific **objects**, such as a database or a specialized processor, or specific **interfaces**, such as a data retrieval interface or a matrix arithmetic interface.

Location Broker software includes the Local Location Broker (LLB), which manages information about resources on the local host; the Global Location Broker (GLB), which manages information about resources available on all hosts; a **client agent** through which programs use the Location Broker facilities; and the `lb_admin` administrative tool.

The GLB stores in a database the locations of objects and interfaces in a network or internet. Clients can use the GLB to access an object or interface, without knowing its location beforehand. The LLB also implements a forwarding facility that provides access by way of a single address to all of the objects and interfaces at the host.

Guide to the Location Broker describes the administration of the Location Brokers.

1.2.3 Network Interface Definition Language Compiler

The NIDL Compiler takes as input an **interface definition** written in NIDL. From this definition, the NIDL Compiler generates client and server **stub** programs. An interface definition specifies the interface between a user of a service and the provider of the service. The definition describes how a client application *sees* a remote service and how a remote server *sees* requests for its service.

The stubs produced by the NIDL Compiler contain nearly all of the *remoteness* in a distributed application. The client stub program performs data conversions, assembles and disassembles packets, and interacts with the RPC run-time library. The server stub program provides similar support for the server. It is easier to write an interface definition in NIDL than it would be to write the stub code that the NIDL Compiler generates from your definition.

1.3 Object Orientation

Programs written with RPC routines access **objects** through interfaces and are cast in terms of the objects they manipulate rather than the machines with which they communicate. Object-oriented programs are easy to design and can readily accommodate changes to hardware and network configurations.

1.3.1 Interfaces, Objects, and Types

An **object** is an entity accessed by welldefined operations. A file, a serial line, a printer, and a processor can all be objects.

Every object has a **type**. Programs can access any object of a given type through one or more **interfaces**, with each interface a set of **operations** that can be applied to any of those objects. For example, you can classify printer queues as objects of the type `printqueue`, accessed through a `directory` interface that includes operations to add, delete, and list jobs in the queues.

As another example of how object, type, and interface apply to distributed applications, consider array processors as objects of the `arrayproc` type. Programs access these objects through either of two interfaces: a `vector` interface, with operations such as `vector$add` and `vector$multiply`, and a `misc` interface, with operations such as `misc$root_mean_square` and `misc$max_abs_val`.

1.3.2 Universal Unique Identifiers

DECrpc identifies every object, type, and interface by a **Universal Unique Identifier** (**UUID**). The UUID is defined as a 16-byte quantity identifying the host on which the UUID is created and the time at which it is created. Six bytes identify the time, two are reserved, and eight identify the host.

The `uuid_gen` utility generates a UUID as a text string or as a data structure defined in C syntax. The string representation used by the Network Interface Definition Language (NIDL) Compiler consists of 28 hexadecimal characters arranged as in this example (the 2 reserved bytes of the 16-byte quantity are not included in the string representation):

```
3a2f883c4000.0d.00.00.fb.40.00.00.00
```

See Section 15.2 for additional information about the structure of a UUID.

1.3.3 Clients and Servers

A **client** is a program that makes remote procedure calls. A remote procedure call requests that a particular operation be performed on a particular object.

A **server** is a program that implements one or more interfaces and provides access to one or more objects. A server accepts requests for operations in any of its interfaces. When it receives a request from a client, it executes the procedures that perform the operation and it sends a response to the client.

All DECrpc applications involve communication between clients and servers through interfaces. However, some applications do not involve specific objects and types. If your application operates on only one object, you can specify `uuid_$nil`, the nil UUID, as the identifier for its type. If your application does not operate on any object, you can specify `uuid_$nil` for both the type and the object.

1.4 Communication Protocols

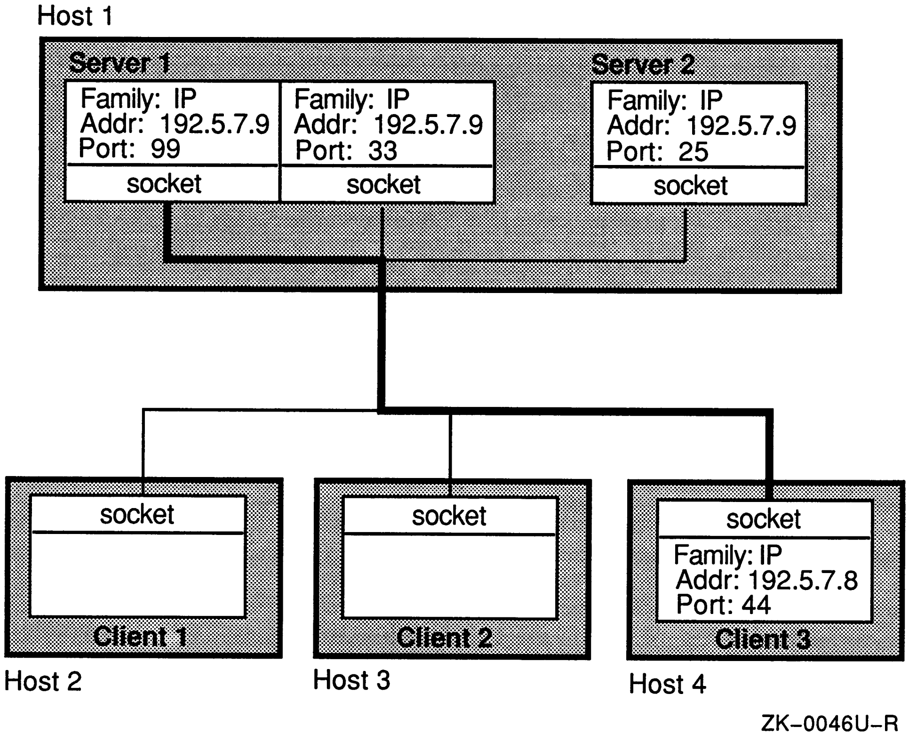
The RPC run-time library is independent of any underlying communications protocol. The DECrpc Version 1.0 run-time library, however, provides support for only the DARPA-defined Internet Protocols (IP).

1.4.1 Sockets and Socket Addresses

The remote procedure calls use the Berkeley UNIX **socket** abstraction for interprocess communications. A socket is an endpoint for communications, in the form of a message queue. An RPC server *listens* on one or more sockets and receives any message delivered to a socket on which it is listening.

Figure 1-1 illustrates RPC communications using sockets. It shows two servers running on one host and several clients on other hosts.

Figure 1-1: RPC Communications



Each socket is identified uniquely by a **socket address**. A socket address, sometimes named `sockaddr`, is a data structure that specifies these socket characteristics:

- Address family
- Network address
- Port number

The **address family**, also called the **protocol family**, determines the communications protocol used to deliver messages and the structure of the addresses used to represent communications endpoints.

The **network address**, given the address family, uniquely identifies a host and contains information sufficient to establish communication with the host. Hosts also have **host IDs**; a host ID uniquely identifies a host but may not be sufficient to establish communication. In the IP family, the network address and the host ID are identical.

The **port number** specifies a communications endpoint within the host. The terms *port* and *socket* are synonymous, but *port number* and *socket address* are not. A port number is one of the three parts in a socket address. For example, the character string *77* might represent a port number, while *ip:wooster[77]* might represent a socket address.

Figure 1-2 illustrates the structure of socket addresses in the IP family.

Figure 1-2: Socket Address Structures

Family	Port	Network Address
16–bit Integer	16–bit Integer	Network ID and Host ID
		32–bit Integer

ZK–0047U–R

A socket address can be represented textually by a string of the form *family:host[port],* where *family* is the textual name of an address family, *host* is either a textual host name or a numeric host ID preceded by a number sign (#), and *port* is a port number. Several of the routines and utilities accept textual representations of socket addresses as input or produce them as output.

Example 1-1 shows two textual representations of socket addresses for the IP address family. The first line shows a textual host name and the second shows a numeric host ID.

Example 1-1: Textual Representations of Socket Addresses

```
ip:cactus[57]
ip:#192.5.7.9[53]
```

1.4.2 Well Known and Opaque Ports

It is possible to design an interface with a specific port number built in. Clients of the interface always send to that port and servers always listen on that port. The port used in such an interface is called a **well known** port. Some well known ports are assigned to particular servers by the administrators of a protocol. For example, the administrators of the Internet Protocols have assigned the port number 23 to the `telnet` remote login facility. All `telnet` servers listen on this well known port, and all `telnet` user programs send to it.

For very widely used services such as `telnet`, well known ports offer a simple way to coordinate communication between clients and servers. For most applications, however, well known ports are impractical. Each protocol family has a limited number of ports, so, unless you obtain an assignment from a central administrator, your application's well known port number is liable to conflict with that of another program.

The Location Broker solves this problem by enabling clients to locate servers without direct use of well known ports. A server can use ports that the RPC run-time library assigns dynamically. The server registers its socket address, including the assigned port, with the Location Broker. A client can then use Location Broker lookup calls to obtain the socket address of the server. The dynamically assigned port is said to be **opaque**, because there is no need for either the client or the server to know the port number.

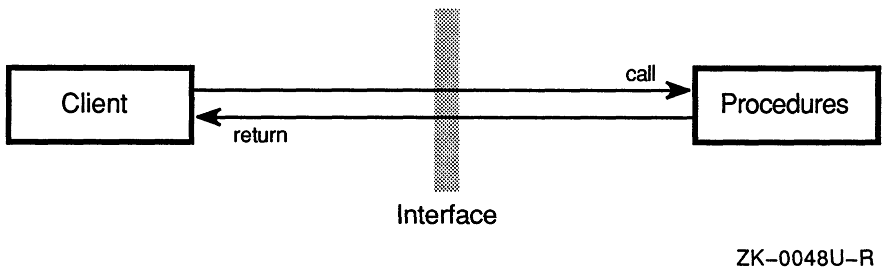
Although the RPC run-time library supports both kinds of ports, if you use opaque ports your application can always coexist with other services.

The Local Location Broker itself uses one well known port to listen for requests. Clients and servers find Global Location Brokers by broadcasting to this port. Section 1.7 describes the Location Broker.

1.5 The Remote Procedure Call Paradigm

Remote procedure calls extend the procedure call mechanism from a single computer to a distributed computing environment. They enable you to distribute the execution of a program among several computers in a way that is transparent to the application code. Figure 1-3 shows the flow of ordinary local procedure calls between a calling client and called procedures.

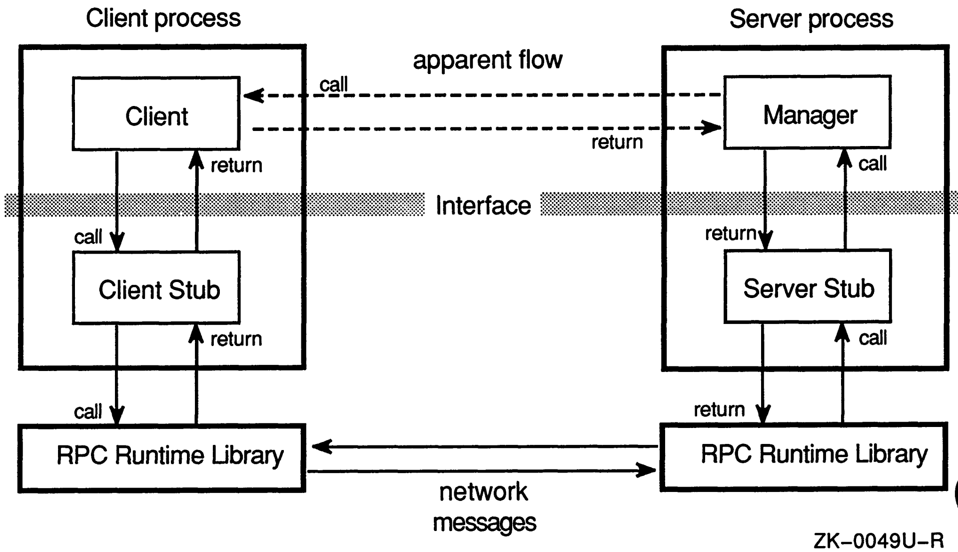
Figure 1-3: Ordinary Local Procedure Call Flow



In contrast to Figure 1-3, which shows the flow of local procedure calls, Figure 1-4 shows the flow of remote procedure calls and illustrates how the RPC paradigm hides the remote aspects of a call from the calling client. The client application uses ordinary calling conventions to request a procedure as if the procedure were a part of the local program, but the procedure is executed by a remote server. The client stub acts as the local representative of the procedure.

A stub is a program module generated by the NIDL Compiler from a user-written interface definition. The stub uses RPC run-time library calls to communicate with the server. Similar activities occur within the server process. Section 1.6 briefly describes interface definitions and Chapter 4 describes the procedure for writing an interface definition.

Figure 1-4: Remote Procedure Call Flow



1.5.1 Interfaces

The interface determines the calling syntax – the signature – for each of its operations. Both client and server procedures use the same syntax. The interface is independent of the mechanism that conveys the request between client and server. It is also independent of the way the operations are implemented. A server that implements the operations in an interface is said to **export** the interface. A client that requests the operations is said to **import** the interface.

For example, suppose that a remote matrix arithmetic package is running as a server on an array processor. Servers on array processor hosts export a `vector` interface containing operations such as `vector$add` and `vector$multiply`. Clients on other hosts import the `vector` interface by calling `vector$add` or `vector$multiply`. The client programs run on their local hosts, but the matrix operations run on the remote array processor.

1.5.2 Clients, Servers, and Managers

An RPC **client** is a program that makes remote procedure calls to request operations. A client does not know how an interface is implemented and might not know the location of a **server** exporting the interface.

An RPC server is a program that performs the operations in one or more interfaces. It executes these operations on objects of one or more types. A server receives requests for operations from clients and it sends responses containing the results of the operations. A server can export interfaces for one object or for several objects. In the array processor example, there is only one object, the array processor. A file server, however, might manage many file objects.

A server can also be a client. For example, a server that gets time from a time server is a client of the time server.

A **manager** is a set of procedures that **implement** the operations in one interface for objects of one type. It is possible for a server to export several interfaces or to export an interface for several types of objects; each combination of interface and type has its own manager.

Figure 1-4 showed the simplest case, a server that exports one interface for objects of one type. Figure 1-5 illustrates a server that exports two interfaces.

Figure 1-5: An RPC Server Exporting Two Interfaces

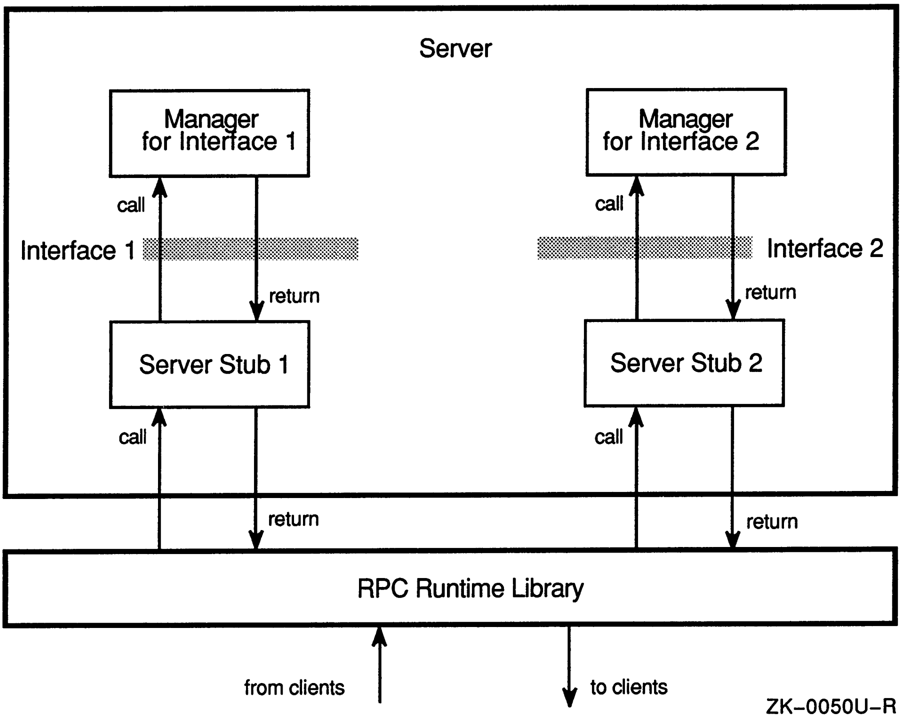
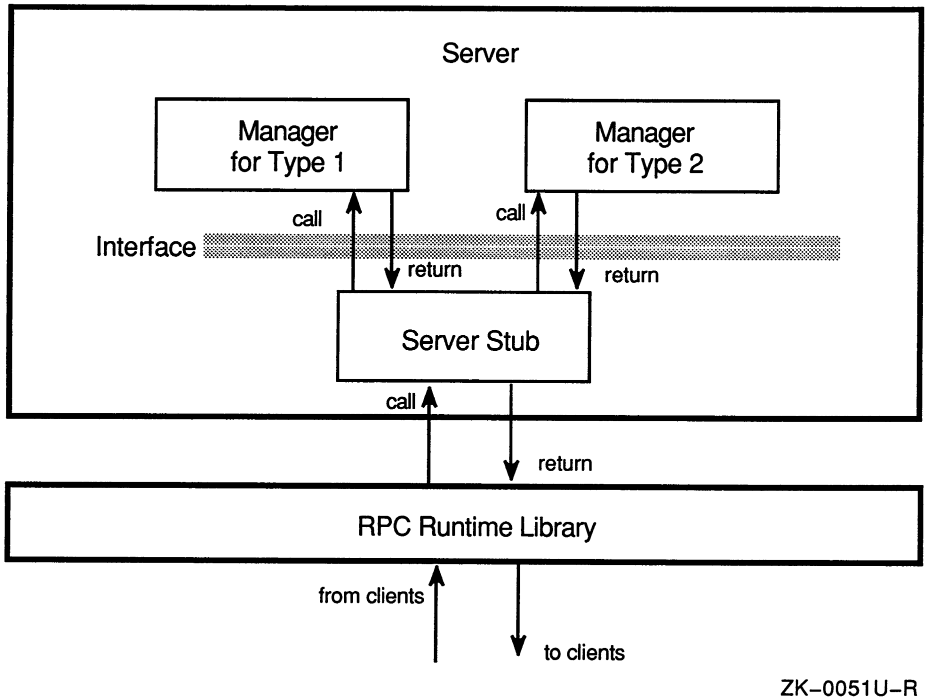


Figure 1-6 shows a server that exports one interface to objects of more than one type.

Figure 1-6: An RPC Server Exporting an Interface for Two Types



1.5.3 Handles

When a client makes a remote procedure call, requesting that a particular operation be performed on a particular object, the RPC run-time library needs the following information to transmit the call:

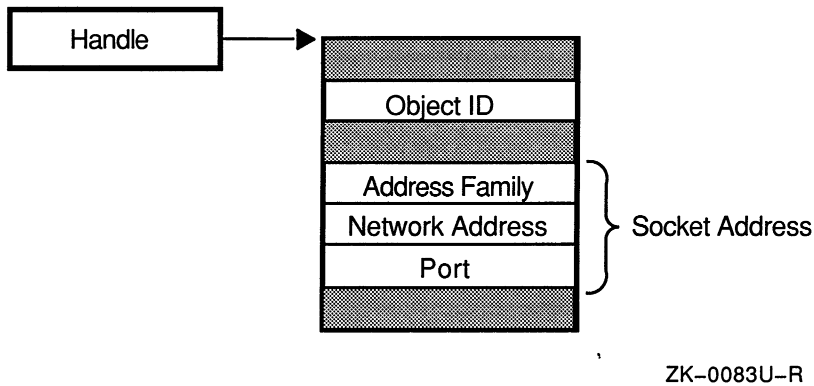
- The object on which the operation is to be performed
- The location of the server that exports the interface containing the operation

The client process represents this information about the object and the server location in a **handle**, which is a pointer to a data structure. The run-time library provides several routines to create and manage handles. Once created, a handle always represents the same object. However, it may represent different servers at different times, or it may not represent a server at all. The server location represented in a handle is called the **binding**. To **bind** a handle is to set its server location.

1.5.3.1 RPC Handles – An RPC handle is a pointer to an opaque data structure containing the information needed to access an object. The name for this pointer type is `handle_t`. In this manual, the term **RPC handle** refers to handle variables of this type and the term **generic handle** refers to handle variables of other types, such as a pathname.

Clients and servers manipulate RPC handles indirectly, through RPC run-time library routines. Figure 1-7 shows an RPC handle.

Figure 1-7: RPC Handle



1.5.3.2 RPC Binding States – An RPC handle can exist in three **binding states:**

unbound

An unbound handle (also called an **allocated** handle) identifies an object but does not identify a location. When a client uses an unbound handle to make a remote procedure call, the RPC run-time library broadcasts the request to all hosts on the local network. Any server that exports the requested interface and supports the requested object can respond. The client accepts the first response it receives. This mechanism is inefficient and has other disadvantages described in Chapter 5.

bound-to-host

A bound-to-host handle identifies an object and a host but does not identify the port number of the server that exports the requested interface. When a client uses a bound-to-host handle to make a remote procedure call, the RPC run-time library sends the request to the host identified in the handle. If the requested interface specifies a well known port, the request goes to that port; otherwise, the request goes to the Local Location Broker forwarding port, and the LLB forwards the request to the server.

fully bound

A fully bound handle (also called a **bound-to-server** handle) identifies an object and the complete socket address of a server. When a client uses a fully bound handle to make a remote procedure call, the RPC run-time library sends the message directly to the socket address identified by the handle.

In all cases, when the client RPC run-time library receives a response from a server, it binds the handle to the server socket address. Therefore, RPC handles are always fully bound when a remote procedure call returns, and the client does not need to use the broadcasting or forwarding mechanism for subsequent calls to the server.

Table 1-1 shows, for each possible binding state of a handle when a remote procedure call is made, the information that the handle represents, the delivery mechanism of the remote procedure call, and the binding state when the procedure call returns.

Table 1-1: RPC Binding States

Binding State on Call	Information Represented	Delivery Mechanism	Binding State on Return
Unbound	Object	Broadcast to all hosts on the local network	Fully bound
Bound-to-host	Object Host	Sent to LLB forwarding port at host	Fully bound
Fully bound	Object Host Server	Sent to specific port at host	Fully bound

1.5.4 Handle Representations and Binding Techniques

DECrpc provides a choice of handle representations and binding techniques. It allows applications to use:

- Explicit or implicit handles
- Manual or automatic binding

The **handle representation**, explicit or implicit, determines whether the client represents handle information with a parameter in each operation or with a global variable. The **binding technique**, manual or automatic, determines whether the client uses RPC handles directly or uses generic handles that are then converted to RPC handles by automatic binding routines. Table 1-2 summarizes the effects of the handle representation and the binding technique on the handle variable.

Table 1-2: Handle Representations and Binding Techniques

Handle	Manual Binding	Automatic Binding
Explicit Handle	Data Type: `handle_t` Representation: Operation parameter	Data Type: Generic, user defined Representation: Operation parameter
Implicit Handle	Data Type: `handle_t` Representation: Client global variable	Data Type: Generic, user defined Representation: Client global variable

1.5.4.1 Explicit and Implicit Handles – In an application that uses **explicit handles**, each operation in the interface must have a handle variable as its first parameter. This parameter passes explicitly from the client to the server, through the client stub, the client and server RPC run-time libraries, and the server stub. (The server run-time library manipulates the location information in the handle so that, on the server side of the application only, the handle specifies the location of the client making the call. The server can thereby identify its client. Of course, the handle always represents the same object.)

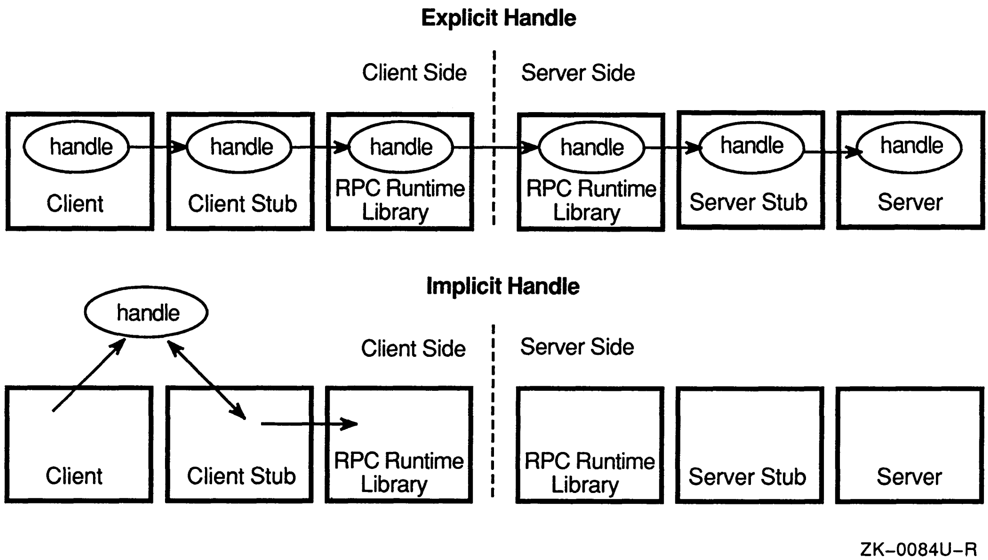
In an application that uses **implicit handles**, the handle identifier is a global variable in the client. The operations do not need to include a handle parameter, and the server does not receive a handle. When the client stub delivers a remote procedure call, it uses the implicit handle variable to supply the handle information needed by the client RPC run-time library.

An implicit handle makes remote procedure calls look more like ordinary procedure calls, because there is no need to pass *special* information in each call. However, this added simplicity comes at the expense of reduced flexibility. Applications that use implicit handles have two major limitations:

- Because the server does not receive the object identifier that a handle contains, the client can access only one object at any time, unless it explicitly passes some other form of object identifier, such as a pathname, as an operation parameter.
- Because all remote procedure calls use the same global variable, the client can access only one server at any time. For example, you cannot use implicit handles in applications that divide computation in parallel among several hosts.

Figure 1-8 illustrates the differences between explicit and implicit handles.

Figure 1-8: Explicit and Implicit Handles



1.5.4.2 Manual and Automatic Binding – In an application that uses **manual binding**, the handle variable is an RPC handle, and the client makes all the RPC run-time library calls that create and bind the handle.

In an application that uses **automatic binding**, the handle variable is generic, and the application developer must supply **autobinding** and **autounbinding** routines that convert generic handles (used by the client) to RPC handles (used by the RPC run-time library). The client stub invokes the autobinding routine each time the client makes a remote procedure call; it invokes the autounbinding routine after the remote call returns. The generic handle variable must contain information sufficient for the autobinding routine to generate an RPC handle.

Automatic binding offers convenience at the expense of performance. Each time the client stub processes a remote procedure call, it must call routines to convert between generic handles and RPC handles. Thus, an interface that uses automatic binding can require more processing than one in which the client performs the binding once and passes an RPC handle to the stub. The difference in performance is smallest in interfaces such as the remote file system example, where each call is likely to require rebinding of the handle.

Table 1-3 shows the differences between manual and automatic binding when a client makes a remote procedure call.

Table 1-3: Comparison of Steps in Manual and Automatic Binding

Manual Binding	Automatic Binding
1. *Client:* Generates RPC handle Binds handle, as necessary Makes procedure call to stub	1. *Client:* Using generic handle, makes procedure call to stub
2. *Client stub:* Sends request to server Receives response from server Returns to client	2. *Client stub:* Calls autobinding routine
3. *Client:* Receives call return from stub Manages RPC handle, as necessary, including unbinding the handle	3. *Autobinding routine:* Generates RPC handle from generic handle Binds RPC handle as necessary Returns RPC handle to stub
	4. *Client stub:* Sends request to server Receives response from server Calls autounbinding routine
	5. *Autounbinding routine:* Frees handles as necessary Returns to stub
	6. *Client stub:* Returns to client
	7. *Client:* Receives call return from stub

Chapter 7 includes an example of an automatic binding routine.

1.5.5 Stubs

Both clients and servers are linked (in the sense of combining object modules to form executable files) with stubs, which are generated by the NIDL Compiler from a user-written interface definition. The client stub takes the place of the remote procedures in the client process and the server stub takes the place of the client in the server process. Stubs make remote procedure calls resemble local calls, which enables clients and servers to use the RPC facilities almost transparently.

The client stub **marshalls** data (copies data into an RPC packet) and **unmarshalls** data (copies data from an RPC packet) and transmits and receives the packet from the server stub.

When a client calls an interface operation, it invokes a routine in the client stub. The client stub then performs these actions:

1. Marshalls the input parameter values
2. Calls `rpc_$sar`, an RPC run-time library routine called only by stubs, to send the packet to the server stub and await a reply
3. Receives the reply packet
4. Unmarshalls the output parameters from the reply packet into the data types expected by the client (that is, the data types specified in the interface definition)
5. Converts the output data to the client's native representation, if the client's native representation is different (for example, converts characters from EBCDIC to ASCII)
6. Returns to the client

Similarly, the RPC run-time library at a server host calls a server stub routine when the server receives a request from the client. The server stub then performs these actions:

1. Unmarshalls the input parameters from the request packet into the data types expected by the server (that is, the data types specified in the interface definition)
2. Converts the input data into the representation native to the server, if the client uses a different representation (for example, converts characters from ASCII to EBCDIC)
3. Calls the manager procedure that implements the operation
4. Marshalls the output parameter values into an RPC packet
5. Returns the packet to the RPC run-time library for transmission to the client stub

As the preceding summary shows, stub procedures in both the client and the server check the data representation format in incoming packets. Each side uses its native format when it marshalls parameters. A label in the header of each transmitted packet indicates the sender's data representation format for integers, characters, and floating-point numbers. If the sender's representation of a data type is different from the receiver's representation, the receiving stub converts that data type when it unmarshalls values.

There is no conversion of data if the sending and receiving hosts have identical representations. This technique allows heterogeneity at minimum cost.

The NIDL Compiler automatically generates source code for the client and server stubs from a definition of the interface written in Network Interface Definition Language. Section 1-6 provides more information about the NIDL Compiler and the stubs that it generates. Chapter 6 describes NIDL syntax in detail.

1.6 Interface Definitions and the NIDL Compiler

An interface definition written in NIDL defines the signatures for each operation in an interface. The NIDL Compiler takes this definition as input and generates C source code files that you can use in building an application.

1.6.1 Interface Definitions

An interface definition describes the constants, types, and operations associated with an interface. NIDL contains constructs for specifying all of this information, but it contains no executable constructs; NIDL is strictly a declarative language. DECrpc supports the C syntax of NIDL and all of the examples in this book are in the C syntax.

Chapter 3 introduces NIDL interface definitions with a simple example and describes the input and output files in the NIDL Compilation. Chapter 4 describes how to write an interface definition and Chapter 6 completely describes the C syntax of NIDL.

1.6.2 Files Generated by the NIDL Compiler

The NIDL Compiler translates a NIDL interface definition into stub modules that you then link with clients and servers. As Section 1.5.5 described, these modules facilitate remote procedure calls by copying arguments to and from RPC packets, converting data representations as necessary, and calling the RPC run-time library.

In addition to stub files, the NIDL Compiler generates C language header files.

1.7 The Location Broker

The Location Broker provides clients with information about the locations of objects and interfaces. Servers register with the Location Broker their socket addresses and the objects and interfaces to which they provide access. Clients issue requests to the Location Broker for the locations of objects and interfaces they wish to access; the broker returns database entries that match an object, type, interface, or combination of these, as specified in the request.

The Location Broker also implements the RPC message-forwarding mechanism. If a client sends a request for an interface to the Location Broker forwarding port on a host, the broker automatically forwards the request to the appropriate server on the host.

This chapter describes the structure and function of the Location Broker software and databases. *Guide to the Location Broker* explains how to configure and administer the Location Brokers.

1.7.1 Location Broker Software

The Location Broker consists of the following interrelated components:

Local Location Broker (LLB)
: The Local Location Broker is a server that maintains a database of information about objects and interfaces located on the local host. The LLB runs as the process `llbd`. The LLB provides access to its database for application programs and also provides the Location Broker forwarding service. An LLB must run on any host that runs DECrpc servers.

Global Location Broker (GLB)
: The Global Location Broker is a server that maintains information about objects and interfaces throughout the network or internet. The GLB process is named `nrglbd`.

Location Broker Client Agent
: The Location Broker Client Agent is a set of library routines that application programs call indirectly to access LLB and GLB databases. When a program issues any Location Broker call, the call goes to the Client Agent at the local host. The Client Agent then performs the actual lookup or update of information in the appropriate Location Broker database.

1.7.2 Location Broker Data

Each entry in a Location Broker database contains information about an object, an interface, and the location of a server that exports the interface to the object. Table 1-4 lists the fields in a database entry.

Table 1-4: Location Broker Database Entry

Field	Description
Object UUID	The unique identifier of the object
Type UUID	The unique identifier that specifies the type of the object
Interface UUID	The unique identifier of the interface to the object
Flag	A flag that indicates whether the object is global (and therefore should be registered in the GLB database)
Annotation	64 characters of user-defined information
Socket address length	The length of the socket address field
Socket address	The location of the server that exports the interface to the object

Because each database entry contains one object UUID, one interface UUID, one type UUID, and one socket address, a Location Broker database must have an entry for each possible combination of object, interface, and socket address. Thus, the database must have 10 entries for a server that:

- Listens on two sockets, `socket_a` and `socket_b`
- Exports `interface_1` for `object_x`, `object_y`, and `object_z`
- Exports `interface_2` for `object_p` and `object_q`
- Has only one `type` UUID

When you look up Location Broker information, you specify any combination of the object UUID, type UUID, and interface UUID as keys, and you request the information from the GLB database or from a particular LLB database. Thus, for example, you can obtain information about all objects of a specific type, all hosts with a specific interface to an object, or all objects and interfaces at a specific host.

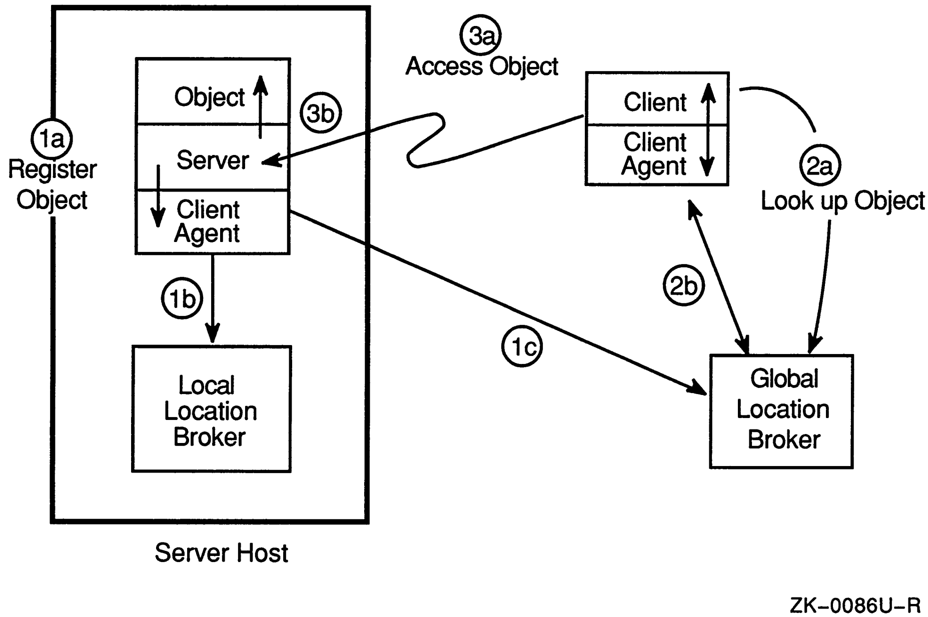
1.7.3 Location Broker Registrations and Lookups

This section describes how servers register their locations with the Location Broker and how clients use Location Broker lookups to locate servers.

Figure 1-9 illustrates a typical case in which a client requires a particular interface to a particular object but does not know the location of a server exporting the interface to the object. In this figure, a server registers itself with the Location Broker by calling the Client Agent in its host (1a). The

Client Agent registers the server with the LLB at the server host (1b) and with the GLB (1c). To locate the server, the client issues a Location Broker lookup call (2a). The Client Agent on the client host sends the lookup request to the GLB, which returns it through the Client Agent to the client (2b). The client can then use RPC calls to communicate directly with the located server (3a, 3b).

Figure 1-9: Client Agents and Location Brokers



1.7.4 The Local Location Broker

The LLB manages information about servers running on the local host. It also acts as a forwarding agent for remote procedure calls.

The forwarding facility of the LLB eliminates the need for a client to know the specific port that a server uses and thereby helps to conserve well known ports. The LLB listens on one well known port per address family. It forwards any messages that it receives to the local server that exports the requested object.

Forwarding is particularly useful when the requestor of a service already knows the host where the server is running. The server can use a dynamically assigned opaque port and register only with the LLB at its local host, not with GLB. To access the server, a client needs to specify the object, the interface, and the host, but not a specific port.

Although it is recommended that you run an `llbd` on every host, the process is required only on hosts that run RPC servers. *Guide to the Location Broker* describes Location Broker configuration and the `lb_admin` utility.

1.7.5 The Global Location Broker

The GLB manages information about servers running anywhere in the network or internet. Clients typically issue lookup calls to the GLB when they do not know at what host a server is running.

Guide to the Location Broker describes how to configure the Global Location Broker.

1.7.6 Designing an Application to Use Global Name Services

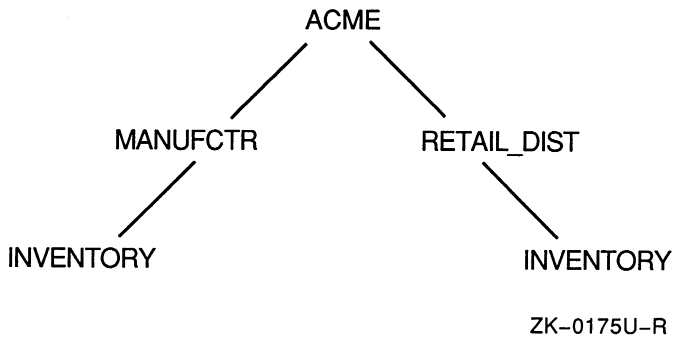
Currently, DECrpc uses the Location Broker as its sole name service. However, when designing an application that may eventually migrate to other environments, you should accommodate the naming requirements of global name services such as Digital Distributed Name Service (DECdns), X.500, and Hesiod/`bind`. Such services use global names to provide a means of advertising and locating computing resources in any size network.

Global names reflect a naming scheme that is distinct from the UUID-based naming scheme of the Location Broker. A global name, like a UUID, is a unique identifier with universal scope. Unlike a UUID, a global name is an easy-to-read, structured text string that is meaningful to users in a particular computing environment. For example, a DECdns global name comprises a series of text strings, read from left to right, that begin with a dot (such as .ACME_CORP.MANUFCTR.INVENTORY). Establishing naming conventions for a given computing environment helps users to specify unique global names.

Being structured enables global names to represent one thing in terms of its relationship to other things. For instance, in a full DECdns global name, each successive string is subordinate to the preceding string. The rightmost string is a **simple name** that identifies a specific resource. For example, the full global name .ACME_CORP.MANUFCTR.INVENTORY reflects the organization of a hypothetical company, Acme Corporation; the first string represents the company as a whole, the middle string represents Acme's manufacturing division, and the final string is a simple name representing a specific account named INVENTORY on a system in the manufacturing division.

All name services maintain a database whose individual entries correspond to a specific resource. Database organization, however, differs between the Location Broker and global name services. In a Location Broker database, each entry has only unstructured identifiers (an object UUID, interface UUID, and/or object type). These unstructured identifiers limit the Location Broker to a flat database, whose entries reside side by side, much like the files of a single-level directory. In contrast, in a global name-service database, each entry has a global name whose textual and structural information dictates the relative placement of the entry in the database.

When entries have full global names, the entries reside in subgroups, much like files in subdirectories within a multiple-level directory. This allows resources belonging to different groups to have the same simple name. For example, the DECdns entries .ACME_CORP.MANUFCTR.INVENTORY and .ACME_CORP.RETAIL_DIST.INVENTORY would reside in a directory tree with the following organization:



The differences in the naming schemes of the Location Broker and global name services can obstruct the eventual migration of a DECrpc application from the Location Broker to a global name service. Though global name services can interpret UUIDs, the exclusive use of UUIDs to identify objects is incompatible with the structural aspects of global naming schemes.

Moreover, the Location Broker can look up an entry by its object type, but some global name services cannot. Therefore, when designing a DECrpc application that might eventually use a global name service, you should constrain the use of Location Broker as follows:

- Avoid proliferating UUIDs as object IDs. You can isolate UUIDs, for example, by restricting them to the `lb_$register` and `lb_$lookup_object` routines or by creating a table to map UUIDs to object names.
- Avoid defining object types.

DECrpc Software 2

DECrpc software includes processes and utilities, library routines, interface definition files, and header files. This chapter provides a survey of the software to give you general background for the programming information in Chapter 4, Writing Interface Definitions; Chapter 5, Developing Distributed Applications; and Chapter 7, Special Topics. Table 2-1 lists the DECrpc software.

Chapters 8 through 16 contain reference pages for each utility, library routine, and process.

Table 2-1: DECrpc Software

Software	Description
`nidl`	Network Interface Definition Language compiler
`uuid_gen`	UUID generating program
`stcode`	Status code translator
`llbd`	Local Location Broker Process
`nrglbd`	Global Location Broker Process (nonreplicatable)
`lb_admin`	Location Broker administrative tool
`.idl` files	Interface definitions
`.h` files	C header files
Library routines	`rpc_$`, `rrpc_$`, `socket_$`, `lb_$`, `uuid_$`, `error_$`, `pfm_$`, and `pgm_$` routines

2.1 Processes and Utilities

The programs described in this section run as VMS foreign commands. The utilities `nidl`, `uuid_gen` and `stcode` help you to develop distributed applications. The Location Broker processes, `nrglbd` and `llbd`, enable client applications to locate servers on remote hosts. The administrative tool, `lb_admin`, helps you to maintain Location Broker databases.

Before running the utilities, define them as foreign commands in the site SYS$MANAGER:SYSLOGIN.COM, as shown in this example:

```
$ lb_admin :== rpc$exe:rpc$lb_admin.exe
$ uuid_gen :== rpc$exe:rpc$uuid_gen.exe
$ nidl :== rpc$exe:rpc$nidl.exe
$ stcode :== rpc$exe:rpc$stcode.exe
```

Chapter 16 includes a reference page for each process and utility.

2.1.1 The uuid_gen Utility

The `uuid_gen` utility generates a UUID. Depending on the qualifiers you specify, `uuid_gen` produces as output a character string representing a UUID, a C initialization for the UUID, or a skeletal interface definition file in the C syntax of NIDL.

2.1.2 The NIDL Compiler

The NIDL Compiler, `nidl`, compiles interface definitions. It takes as input an interface definition written in NIDL. It produces as output a server stub, a client stub, and a client switch (all in the C language), together with header files.

2.1.3 Location Broker Processes

DECrpc includes processes that manage the Local Location Broker (LLB) database and the Global Location Broker (GLB) database.

Any host that runs an RPC server must also run the LLB process, `llbd`. Any network that supports RPC activity must have at least one host running a GLB process. In an internet, at least one GLB process must run in each network.

The Location Broker processes typically run as detached processes. On most VMS systems, they start at boot time from the file SYS$STARTUP:RPC$UCX_STARTUP.COM.

See *Guide to the Location Broker* for more information on Location Broker configuration.

2.1.4 Location Broker Administrative Tool

The `lb_admin` utility allows you to inspect or modify the contents of a Location Broker database. It provides lookup, register, unregister, and cleanup operations. It can perform these operations on any LLB or GLB database.

2.1.5 Status Code Translator

The `stcode` utility translates hexadecimal status code values produced by programs to textual messages.

2.2 The rpc_$ Client and Server Library Routines

The `rpc_$` library routines constitute the interface to the RPC run-time library. Some of these routines are used only by clients, some only by servers, and some by either clients or servers.

The following sections describe each set of routines.

Chapter 12 includes a reference page for each `rpc_$` routine.

2.2.1 Client Routines

Most of the `rpc_$` client routines either create a handle or manage its binding state.

`rpc_$alloc_handle`

> Allocates an RPC handle that identifies a specific object but not a specific server.

`rpc_$set_binding`

> Sets the binding in an allocated handle so that it specifies a socket address.

`rpc_$bind`

> Allocates an RPC handle and sets its binding. This call has the same effect as an `rpc_$alloc_handle` call followed by an `rpc_$set_binding` call.

`rpc_$clear_server_binding`

> Removes the association of an RPC handle with a server, but retains the association with a host. If a client uses this handle to make a remote procedure call, the call is sent either to a well known port or to the Local Location Broker forwarding port on the remote host.

`rpc_$clear_binding`

> Removes the association of an RPC handle with a server and a host. This call saves the handle for reuse in accessing the same object, possibly via a different server. If a client uses this handle to make a remote procedure call, the call is broadcast.

`rpc_$dup_handle`

> Returns a copy of an existing RPC handle. A handle is not freed until `rpc_$free_handle` is called on all copies of the handle.

`rpc_$free_handle`

Frees an RPC handle. This call removes any association of the handle with an object and an address and releases the handle.

`rpc_$set_async_ack`

Sets or clears asynchronous-acknowledgement mode in a client. Asynchronous-acknowledgement mode allows a client to acknowledge its receipt of replies from servers asynchronously, for greater efficiency.

`rpc_$set_short_timeout`

Sets or clears short-timeout mode on a handle. If a client uses a handle in short-timeout mode to make a remote procedure call, but the server shows no signs of life, the call fails quickly.

`rpc_$sar`

Sends a remote procedure call request and awaits a reply from the server. This call is for use only by client stubs that the NIDL Compiler generates, so there is no reference description for it.

2.2.2 Server Routines

This section describes the `rpc_$` server routines, most of which initialize the server so that it has a socket on which to listen and is registered with the RPC run-time library on its host.

`rpc_$use_family`

Creates a socket that the server will use to communicate with clients. You specify the address family. The run-time library assigns an available port number for the socket.

`rpc_$use_family_wk`

Creates a socket that uses a well known port. You specify both the address family and the port number.

`rpc_$register`

Registers an interface with the RPC run-time library. This call is superseded by `rpc_$register_mgr` and `rpc_$register_object`. Any server that contains more than one implementation of a type interface or more than one version of a manager must use `rpc_$register_mgr` rather than `rpc_$register`.

`rpc_$register_mgr`

Registers a generic interface with the RPC run-time library. You specify an interface, a type for which the server exports the interface, and the set of manager procedures that implement the interface for that type. Any server that contains more than one implementation or more than one version of a manager must use this call rather than `rpc_$register`.

`rpc_$register_object`

Registers an object with the RPC run-time library. You declare an object for which the server exports interfaces and declare the type of the object.

`rpc_$unregister`

Unregisters an interface that was previously registered with the server by the `rpc_$register_mgr` or `rpc_$register` routines. The server will not respond to requests for the unregistered interface.

`rpc_$listen`

Listens for remote procedure call requests from clients. When a request is received, call the requested manager procedure for the requested operation and send the result in a reply to the client.

`rpc_$inq_object`

Returns the UUID of the object represented by an RPC handle. This call enables manager procedures to determine the specific object that they must access.

`rpc_$shutdown`

Shuts down. The server stops processing incoming requests and `rpc_$listen` returns.

`rpc_$allow_remote_shutdown`

Allows or disallows remote shutdown initiated by `rpc_$shutdown`.

`rpc_$set_fault_mode`

Controls handling of faults that occur in server routines. By default, the server reflects faults back to the client and continues processing. You can use this routine to set the fault-handling mode so that the server sends a ‘‘communications failure’’ fault to the client and exits.

2.2.3 Routines for Clients or Servers

The `rpc_$` routines listed in this section can be used by both clients and servers.

`rpc_$inq_binding`

Returns the socket address identified by an RPC handle. Typically, a client uses this call to identify the specific server that responded to a remote procedure call.

rpc_$inq_object

Returns the UUID of the object represented by an RPC handle.

rpc_$name_to_sockaddr

Given a host name and port number, returns the equivalent socket address. This call is superseded by socket_$from_name.

rpc_$sockaddr_to_name

Given a socket address, returns the equivalent host name and port number. This call is superseded by socket_$to_name.

2.3 The rrpc_$ Client Library Routines

This section describes the rrpc_$ routines. These routines enable a client to request information about a server or to shut down a server.

Chapter 13 includes a reference page for each rrpc_$ routine.

rrpc_$are_you_there

Checks whether a server is answering requests.

rrpc_$inq_stats

Obtains statistics about a server.

rrpc_$inq_interfaces

Obtains a list of the interfaces that a server exports.

rrpc_$shutdown

Shuts down a server, if the server allows it. See also rpc_$allow_remote_shutdown.

2.4 The socket_$ Library Routines

This section describes the socket_$ routines. These routines manipulate socket addresses. Unlike the calls that operating systems typically provide, the socket_$ routines operate on addresses of any protocol family.

Chapter 14 includes a reference page for each socket_$ routine.

socket_$equal

Compares two socket addresses.

socket_$to_name

Converts a socket address to a textual host name and port number.

socket_$to_numeric_name

Converts a socket address to a numeric host name and port number.

`socket_$from_name`

Converts a textual host name and port number to a socket address.

`socket_$family_to_name`

Converts the integer value of a protocol family to its textual name.

`socket_$family_from_name`

Converts the textual name of a protocol family to its integer value.

`socket_$valid_family`

Checks whether an address family is usable.

`socket_$valid_families`

Lists the address families that are usable.

2.5 The lb_$ Library Routines

This section describes the `lb_$` routines. These routines constitute the interface to the Location Broker Client Agent. The routines direct the Client Agent to look up, register, or unregister entries in a Location Broker database.

Chapter 9 includes a reference page for each `lb_$` routine.

`lb_$lookup_object`

Finds entries in the GLB database that match the specified object identifier.

`lb_$lookup_type`

Finds entries in the GLB database that match the specified type identifier.

`lb_$lookup_interface`

Finds entries in the GLB database that match the specified interface identifier.

`lb_$lookup_object_local`

Finds entries in the specified LLB database that match the specified object identifier.

`lb_$lookup_range`

Finds entries in the specified database (LLB or GLB) that match the specified combination of object, type, and interface UUIDs.

`lb_$register`

Registers a specific object and interface, that is, creates an entry in the Location Broker database. You can specify an entry as local or global. If it is local, it will be registered only in the LLB. If it is global, it will also be registered in the GLB.

`lb_$unregister`

Unregisters a specific object and interface, that is, removes an entry from the Location Broker database.

2.6 The uuid_$ Library Routines

This section describes the `uuid_$` routines. These routines generate and manipulate Universal Unique Identifiers.

Chapter 15 includes a reference page for each `uuid_$` routine.

`uuid_$gen`

Generates a new UUID.

`uuid_$decode`

Converts a character-string representation of a UUID (as generated by the `uuid_gen` utility) into a `uuid_$t` value that is usable by a program.

`uuid_$encode`

Converts a UUID into its character-string representation.

`uuid_$equal`

Compares two UUIDs.

2.7 The error_$ Library Routines

Most of the run-time library routines indicate their completion status with status codes. The `error_$` routines, which are listed in this section, convert these status codes into textual error messages.

Chapter 8 includes a reference page for each `error_$` routine.

`error_$c_get_text`

Returns system, module, and error texts for a status code.

`error_$c_text`

Returns an error message for a status code.

2.8 The pfm_$ Library Routines

The `pfm` fault management routines, which are described in this section, allow programs to manage signals, faults, and exceptions by establishing cleanup handlers.

Chapter 10 includes a reference page for each `pfm_$` routine.

`pfm_$cleanup`

Establishes a cleanup handler.

`pfm_$enable`

Enables asynchronous faults after they have been inhibited by a call to `pfm_$inhibit`.

`pfm_$enable_faults`

Enables asynchronous faults after they have been inhibited by a call to `pfm_$inhibit_faults`.

`pfm_$inhibit`

Inhibits asynchronous faults.

`pfm_$inhibit_faults`

Inhibits asynchronous faults but allows task switching.

`pfm_$init`

Initializes the PFM package.

`pfm_$reset_cleanup`

Resets a cleanup handler.

`pfm_$rls_cleanup`

Releases cleanup handlers.

`pfm_$signal`

Signals the calling process.

2.9 The pgm_$ Library Routine

The `pgm_$exit` program management routine is often used at the end of a cleanup handler to terminate a program.

Chapter 11 includes a reference page for the `pgm_$exit` routine.

`pgm_$exit`

Exits from the calling program.

2.10 The System IDL Directory

The system IDL directory, RPC$IDL, contains several interface definition files distributed with DECrpc.

2.10.1 Interface Definition Files for Types and Constants

The following files in the system IDL directory define only data types and constants, not operations:

`RPC$IDL:BASE.IDL`

Defines some basic types and constants.

`RPC$IDL:NBASE.IDL`

Defines types and constants used in network interfaces.

`RPC$IDL:NCASTAT.IDL`

Defines the completion status codes specified by the RPC run-time library.

Several of the interface definitions described in the following sections import one or more of these files.

2.10.2 Interface Definition Files for Local Interfaces

The following files in the system IDL directory define local interfaces:

`RPC$IDL:LB.IDL`

Defines the interface to the Location Broker Client Agent.

`RPC$IDL:RPC.IDL`

Defines the interface to the RPC run-time library. The NIDL Compiler automatically imports `RPC$IDL:RPC.IDL` when it compiles the definition for any remote interface.

`RPC$IDL:SOCKET.IDL`

Defines types, constants, and operations pertaining to socket addresses and protocol families.

`RPC$IDL:UUID.IDL`

Defines types, constants, and operations pertaining to UUIDs

The operations in these interfaces cannot be called remotely. The NIDL defines the interfaces so that header files can be generated from a common source. The NIDL files, rather than the generated header files, serve as readable descriptions of the interfaces.

2.10.3 Interface Definition Files for Remote Interfaces

The following files in the system IDL directory define remote interfaces:

RPC$IDL:CONV.IDL

Defines operations that manage client-server conversations.

RPC$IDL:GLB.IDL

Defines the interface to the Global Location Broker.

RPC$IDL:LLB.IDL

Defines the interface to the Local Location Broker.

RPC$IDL:RRPC.IDL

Defines operations that a client can use to request information about a server or to shut down a server.

You do not ordinarily need to call any of the operations in the `conv_`, `glb_`, and `llb_` interfaces, because you can access most of their functionality through the `lb_` and `rpc_` interfaces.

The `rrpc_` interface is automatically exported by every RPC server. Its operations are implemented by the run-time support for the server and are not part of the server proper.

2.11 Header Files and Insert Files

For each of the interface definition files described in the previous section, DECrpc provides corresponding header files in C. DECrpc also provides two header files that are hand coded, not generated from an interface definition.

The C header files reside in SYS$SYSROOT:[RPC$INCLUDE], which is pointed to by the logical RPC$INCLUDE. Many C compilers support qualifiers that allow you to specify this directory as a place for the compiler to look for header files.

RPC$INCLUDE:IDL_BASE.H

This file defines primitives that are present in NIDL but lacking in C, such as the `boolean` type. The `RPC$INCLUDE:IDL_BASE.H` file also contains declarations or definitions for data types, external functions, and macros used by stubs.

RPC$INCLUDE:PFM.H

This file defines a portable interface to the Process Fault Manager subsystem.

Steps in Building a Distributed Application 3

To build a distributed application, you combine code that the NIDL Compiler generates with code that you write. This chapter describes the `binop` application to introduce the steps in building a distributed application. Section 3.1 uses `binop` to illustrate NIDL interface definitions, and Section 3.2 describes the user-written files for the application. Many details, however, are not explained in this section. Chapters 4, 5, 6, and 7 describe interface definition and application development more thoroughly.

See also Section 3.3, which describes `binop_lu`, an application that uses the Location Broker.

3.1 A Distributed Application: The binop Interface Definition

This section describes `binop`, an application that performs integer additions on a remote server. The examples directory, `SYS$SYSROOT:[SYSHLP.EXAMPLES]`, contains the source code files for `binop`.

The `binop` application uses explicit handles and manual binding. The `binop.idl` file, shown in Example 3-1, defines the `binop` interface. Section 3.2 describes the `binop` client and server programs.

Chapter 4 describes how to generate the UUID and the skeletal interface definition file with `uuid_gen`.

Example 3-1: The binop.idl Interface Definition

```
%c   [1]
[uuid(41979f30a000.0d.00.00.fb.40.00.00.00),  [2]
     port( ip:[6677]),version(1)]
interface binop
{

[idempotent]    [3]
void binop$add( [4]
     handle_t [in] h,
     long [in] a,
     long [in] b,
     long [out] *c
     );
}
```

[1] The first line of the interface definition states that the definition uses the C syntax of NIDL.

[2] The next three lines specify the UUID, well known ports, version, and name of the interface.

[3] This operation has the **idempotent** attribute, which specifies that the operation can safely be executed more than once and allows the RPC run-time library to employ more efficient calling semantics.

[4] The remainder of the definition defines the signature of `binop$add`, the one operation in the interface. The first parameter is an RPC handle. The next two are inputs. The last parameter is an output.

Since the `binop` example imports no other interface definitions, defines no constants, and uses only predefined data types, it does not illustrate the NIDL import, constant, and type declarations. Examples in Chapters 5 and 6 illustrate these constructs.

To keep the client and server for `binop` simple, the interface definition specifies well known ports. However, as Chapter 1 recommends, you should avoid well known ports in real applications and use opaque ports instead. Section 3.3 describes the `binop_lu` example, which uses opaque ports by means of Location Broker lookups. Chapters 4 and 5 develop the `binop_fw` example, which uses opaque ports by means of Location Broker forwarding.

To compile an interface definition, run the NIDL Compiler in the examples directory as shown for `binop` in this example:

```
$ nidl binop.idl -m
```

Note

This tool is available on VMS systems, ULTRIX systems, and other versions of the UNIX operating system. The command interface is common across all these systems, and therefore is not in a traditional DCL style.

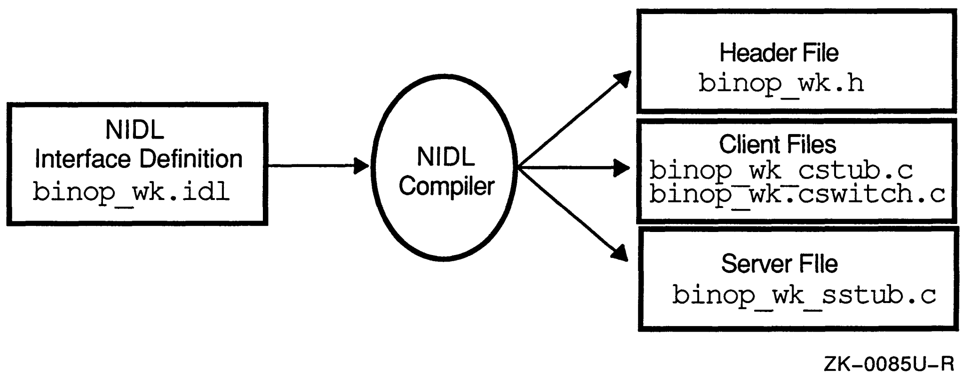
For this command, precede qualifiers with a hyphen (-), rather than the customary slash (/).

You must define each DECrpc command as a foreign command.

The `-m` qualifier allows a server to export more than one version of an interface and to implement an interface for more than one type. The compiler appends the version number to the interface name when it generates identifiers in the stub and header files. For example, the interface specifier for version 2 of the `binop` interface would be `binop_v3$if_spec`.

Figure 3-1 shows the input and output files involved in the compilation of `binop.idl`. If you build the `binop` programs in the example directories, as described in the next section, you can examine the stub, switch, and header files that the NIDL Compiler produces. (The switch file (`binop_cswtch.c`) is generated but is not used.)

Figure 3-1: Input and Output Files in the binop.idl Compilation



The header file `binop.h` declares the `binop$add` procedure, initializes the `binop_v1$if_spec` interface specifier, and defines the `binop_v1$epv_t` data type. It also contains directives to include the standard DECrpc header files that define basic data types and declare RPC run-time library routines.

An `if_spec` is a data structure that clients and servers pass to the RPC run-time library when they bind or register an interface. An `epv_t` is the data type for an **entry point vector** (EPV), a record of pointers to the operations in an interface. If you run the NIDL Compiler with the -m qualifier, which allows multiple versions of an interface, the NIDL Compiler appends the version number, for example, _v1, to the interface name when it generates EPV identifiers.

The binop_cstub.c and binop_cswtch.c modules together implement the client stub. They contain a procedure named binop$add. This procedure marshalls its two input arguments, *a* and *b*, into an RPC packet and calls rpc_$sar to send a remote procedure call. When rpc_$sar returns, the result is unmarshalled from the returned packet into the output argument, *c*.

The module binop_sstub.c is the server stub. It unmarshalls *a* and *b* from the packets sent by clients, then passes those values to the manager procedure binop$add. It marshalls the result, *c*, into an RPC packet and returns control to the RPC run-time library, which sends the packet back to the waiting client.

As Figure 3-1 shows, the NIDL Compiler generates two client files for an interface: a stub file, *interface*_cstub.c, and a switch file, *interface*_cswtch.c. The client switch contains public procedures (such as binop$add), while the client stub contains only private procedures whose names are not visible outside of the _cstub.c file. The client stub defines an EPV containing function pointers to the private procedures, and the client switch invokes these procedures through the EPV (for example, by calling binop_v1$client_epv.binop$add).

To build a client, you link both the client switch and the client stub with the client. The client calls the procedures by their ordinary public names, as specified in the NIDL definition. These procedures are contained in the client switch, which then calls the client stub procedures through the client EPV.

3.2 A Distributed Application: The binop User-Written Files

Section 3.1 described the compiler-generated files for the binop example. This section describes the user-written files:

client.c The main client module

server.c The main server module

binop.c The manager

The client.c and server.c examples shown in this chapter omit some conditional and diagnostic code, the utility module util.c, and the compiler-generated header, stub, and switch files. The binop example directory contains complete source code for binop.

3.2.1 The Client

The `binop` application uses well known ports, explicit handles, and manual binding. The client code generates and binds an RPC handle that it passes as the first argument in its remote procedure calls.

Example 3-2 shows the client module, `client.c`.

Example 3-2: The client.c Module for binop

```
#include <sys/time.h>
#include <stdio.h>
#include "binop.h"
#include "socket.h"
#define CALLS_PER_PASS 100

globalref uuid_$t uuid_$nil;

main(argc, argv)
int argc;
char *argv[];
{
    handle_t h;
    status_$t st;
    socket_$addr_t loc;
    unsigned long llen;
    long i, n;
    int k, passes;
    int start_time, stop_time;

    if (argc != 3) {
        fprintf(stderr, "usage: client hostname passes\n"); 1
        exit(1);
    }

    passes = atoi(argv[2]);

    socket_$from_name(socket_$unspec, (ndr_$char *) argv[1], 2
        (long) strlen(argv[1]), (long) rpc_$unbound_port,
        &loc, &llen, &st);

    h = rpc_$bind(&uuid_$nil, &loc, llen, &st);     3

    for (k = 1; k <= passes; k++) {
        start_time = time(NULL);
        for (i = 1; i <= CALLS_PER_PASS; i++) {
            binop$add(h, i, i, &n); 4
            if (n != i+i)
                printf("Two times %ld is NOT %ld\n", i, n);
        }
        stop_time = time(NULL);
        printf("pass %3d; real/call: %2d ms\n",
            k, ((stop_time - start_time) * 1000) / CALLS_PER_PASS);
    }
}
```

[1] This program takes two arguments: the network address of a host where a server is running and the number of passes to execute.

[2] To convert the network address of the server host to a socket address, the client calls `socket_$from_name`, part of the socket address manipulation interface in the RPC run-time library. Because the port parameter for `socket_$from_name` is the predefined constant `rpc_$unbound_port`, the resulting socket address specifies a host, but not a particular port at that host.

[3] The client then supplies this socket address to the `rpc_$bind` library call, which creates an RPC handle and binds this handle to the socket address. Because the socket address does not specify a port, the `rpc_$bind` call generates a bound-to-host handle. The first argument to `rpc_$bind`, the object identifier, is `uuid_$nil`, because `binop` does not operate on any particular object.

[4] When the client issues its first call to `binop$add`, the RPC run-time library at the client host extracts the well known port number for the server from the `binop_v1$if_spec` interface specifier, so that the handle is fully bound when the run-time library sends the request. The handle remains fully bound for all subsequent calls.

3.2.2 The Server

Example 3-3 shows the server module, `server.c`, which creates a socket, registers with the run-time library at its host, and then listens for a call for its services.

Example 3-3: The server.c Module for binop

```
#include <stdio.h>
#include "binop.h"
#include "socket.h"

globalref uuid_$t uuid_$nil;
globalref binop_v1$epv_t binop_v1$manager_epv;

main(argc, argv)
int argc;
char *argv[];
{
    status_$t st;
    socket_$addr_t loc;
    unsigned long llen;
    unsigned long family;
    socket_$string_t name;
    unsigned long namelen = sizeof(name);
    unsigned long port;

    if (argc != 2) {
```

Example 3-3: (continued)

```
        fprintf(stderr, "usage: server family\n"); [1]
        exit(1);
    }

    family = socket_$family_from_name((ndr_$char *) argv[1], [2]
        (long) strlen(argv[1]), &st);
    rpc_$use_family_wk(family, &binop_v1$if_spec, [3]
        &loc, &llen, &st);

    rpc_$register_mgr( [4]
        &uuid_$nil, [5]
        &binop_v1$if_spec,
        binop_v1$server_epv,
        (rpc_$mgr_epv_t) &binop_v1$manager_epv,
        &st);

    socket_$to_name(&loc, llen, name, &namelen, &port, &st); [6]
    name[namelen] = 0;
    printf("Registered: name='%s', port=%ld\n", name, port);

    rpc_$listen((long) 1, &st); [7]
}
```

[1] This program takes one argument, the name of an address family.

[2] The call to `socket_$family_from_name` converts the address family name into the integer representation of the family, returned as `family`.

[3] The server then supplies `family` and the `binop` interface specifier to `rpc_$use_family_wk`, which creates a socket for the server at its well known port. The `loc` variable stores this socket address.

[4] In order to communicate with clients, a server registers itself with the RPC run-time library at its host. The `binop` server calls `rpc_$register_mgr` to tell the run-time library that it exports the `binop_v1` interface.

[5] The first argument to `rpc_$register_mgr`, the type identifier, is `uuid_$nil`, because `binop` does not operate on an object. If a server operates on objects of several types, as in Figure 1-7, it registers its managers by calling `rpc_$register_mgr` once for each type, and it must register its objects by calling `rpc_$register_object` once for each object.

[6] After registering with the RPC run-time library, the `binop` server calls `socket_$to_name` to extract a textual network address and a

port number from its socket address, and it uses this information to print an announcement of its registration.

7 Finally, it invokes `rpc_$listen` to begin handling remote procedure calls. The first argument to `rpc_$listen` must be 1.

3.2.3 The Manager

The manager module `binop.c`, which is shown in Example 3-4, contains the implementation of the `binop$add` procedure. This code is linked with the server module.

Example 3-4: The binop.c Manager Module

```
#include "binop.h"

globaldef binop_v1$epv_t binop_v1$manager_epv  {  [1]
    binop$add
};

void binop$add(h, a, b, c) [2]
handle_t h;
long a, b, *c;
{
    *c = a + b;
}
```

1 The module first defines the manager EPV `binop_v1$manager_epv`.

2 The next lines contain the actual implementation of the `binop$add` procedure.

3.2.4 Building and Running the binop Application

The `binop` client program is the result of compiling these modules:

- `client.c`
- `util.c`
- `binop_cstub.c`
- `binop_cswtch.c`

The server program is the result of compiling these modules:

- `server.c`
- `util.c`
- `binop.c`
- `binop_sstub.c`

All of these modules contain a `#include` directive to incorporate the definitions in `binop.h`.

To create the `binop` application on your system, execute the `BUILD.COM` file in the example directory. The file runs the NIDL Compiler to generate stub, switch, and header files, and then runs a C compiler to build the client and server programs.

To run `binop`, first define the server and client programs as foreign commands. Then start the server, specifying the `ip` address family:

```
$ server ip
Registered: name'ip:elektra', port=6677
```

After the server has registered itself, run the client, specifying the network address of the server host (in this example, elektra) and the number of passes to execute:

```
$ client ip:elektra 4
pass   1; real/call: 20 ms
pass   2; real/call: 20 ms
pass   3; real/call: 10 ms
pass   4; real/call: 10 ms
```

3.3 Using Location Broker Lookups: The binop_lu Application

Sections 3.1 and 3.2 described `binop`, an application that uses well known ports to coordinate communication between client and server. The examples in this section show a modified version of `binop` that uses opaque ports by means of Location Broker lookups. The modified application is called `binop_lu`. As in the `binop` example, the code shown omits some conditional and diagnostic code.

See also Chapter 1, which describes issues to consider if you are designing an application that in the future may use a name service other than the Location Broker.

3.3.1 The Interface Definition

The interface definition for `binop_lu` (Example 3-5) differs from the definition for `binop` (Example 3-1) in the interface UUID, the interface and operation names, and the absence of well known ports.

Example 3-5: The binop_lu.idl Interface Definition

```
%c
[uuid(41979f38d000.0d.00.00.fb.40.00.00.00), version(1)]
interface binop_lu
{

[idempotent]
void binop_lu$add(
    handle_t [in] h,
    long [in] a,
    long [in] b,
    long [out] *c
    );
}
```

3.3.2 The Client

Example 3-6 contains the code for the binop_lu client. Unlike the binop client, which converts a host name to a socket address, the binop_lu client looks up a server address in the Location Broker database.

Example 3-6: The client.c Module for binop_lu

```
#include <sys/time.h>
#include <stdio.h>
#include "binop_lu.h"
#include "lb.h"
#include "socket.h"
#define CALLS_PER_PASS 100

globalref uuid_$t uuid_$nil;

main(argc, argv)
int argc;
char *argv[];
{
    handle_t h;
    status_$t st;
    lb_$entry_t entry;
    lb_$lookup_handle_t ehandle = lb_$default_lookup_handle;
    unsigned long nresults;
    socket_$addr_t loc;
    unsigned long llen;
    long i, n;
    int k, passes
    int start_time, stop_time;
    if (argc != 2) {
        fprintf(stderr, "usage: client passes\n");  [1]
        exit(1);
    }
    passes = atoi(argv[1]);
```

Example 3-6: (continued)

```
    lb_$lookup_interface(&binop_lu_v1$if_spec.id, &ehandle, 1, [2]
        &nresults, &entry, &st);

    h = rpc_$bind(&uuid_$nil, &entry.saddr, entry.saddr_len, &st); [3]

    for (k = 1; k <= passes; k++) {
        start_time = time(NULL);
        for (i = 1; i <= CALLS_PER_PASS; i++) {
            binop_lu$add(h, i, i, &n);
            if (n != i+i)
                printf("Two times %ld is NOT %ld\n", i, n);
        }
        stop_time = time(NULL);
        printf("pass %3d; real/call: %2d ms\n",
            k, ((stop_time - start_time) * 1000) / CALLS_PER_PASS);
    }
}
```

[1] The binop_lu client program takes only one argument, the number of passes to execute. There is no need for the user to specify a host name.

[2] The lb_$lookup_interface call takes the place of the socket_$from_name call in the binop client (Example 3-2). This lookup call returns a Global Location Broker database entry that matches the binop_lu interface UUID. The returned entry contains, in its saddr field, the socket address of the server.

[3] Addresses in Location Broker entries always specify a port number, so the handle returned by rpc_$bind in this example is fully bound.

3.3.3 The Server

The binop_lu server (Example 3-7) differs from the binop server (Example 3-3) in two important ways:

- The binop_lu server calls rpc_$use_family rather than rpc_$use_family_wk to obtain the socket on which it listens. This call requests the RPC run-time library to dynamically assign an available port.
- The server calls lb_$register to register its interface and its socket address with the Global Location Broker.

Example 3-7: The server.c Module for binop_lu

```
#include <stdio.h>
#include "binop_lu.h"
#include "lb.h"
#include "socket.h"

globalref uuid_$t uuid_$nil;
globalref binop_lu_v1$epv_t binop_lu_v1$manager_epv;

main(argc, argv)
int argc;
char *argv[];
{
    status_$t st;
    socket_$addr_t loc;
    unsigned long llen;
    unsigned long family;
    socket_$string_t name;
    unsigned long namelen = sizeof(name);
    unsigned long port;
    lb_$entry_t entry;

    if (argc != 2) {
        fprintf(stderr, "usage: server family\n");
        exit(1);
    }

    family = socket_$family_from_name((ndr_$char *) argv[1],
        (long) strlen(argv[1]), &st);
    rpc_$use_family(family, &loc, &llen, &st); [1]

    rpc_$register_mgr(
        &uuid_$nil,
        &binop_lu_v1$if_spec,
        binop_lu_v1$server_epv,
        (rpc_$mgr_epv_t) &binop_lu_v1$manager_epv,
        &st);

    lb_$register(&uuid_$nil, &uuid_$nil, &binop_lu_v1$if_spec.id, 0, [2]
        (ndr_$char *) "binop_lu example", &loc, llen, &entry, &st);

    socket_$to_name(&loc, llen, name, &namelen, &port, &st);
    name[namelen] = 0;
    printf("Registered: name'%s', port=%ld\n", name, port);

    rpc_$listen((long) 1, &st);
}
```

[1] The call to `rpc_$use_family` requests the RPC run-time library to dynamically assign an available port.

[2] The call to `lb_$register` registers the interface and its socket address with the Global Location Broker. The first two arguments to

lb_$register, the object and type identifiers, are both uuid_$nil, because binop does not operate on an object. The server supplies the text string "binop_lu example" as an annotation for its Location Broker database entry.

3.3.4 The Manager

Except for name changes, the binop_lu manager (Example 3-8) is the same as its counterpart in binop (Example 3-4).

Example 3-8: The binop_lu.c Manager Module

```
#include "binop_lu.h"

globaldef binop_lu_v1$epv_t binop_lu_v1$manager_epv  {
    binop_lu$add
};

void binop_lu$add(h, a, b, c)
handle_t h;
long a, b, *c;
{
    *c = a + b;
}
```

3.3.5 Building and Running the binop_lu Application

You must set up Location Broker services on your network or internet before you can run the binop_lu client and server. A Global Location Broker should be running on at least one host in the network or internet where you intend to run a client or server. A Local Location Broker should be running on each host where you intend to run a server. *Guide to the Location Broker* contains guidelines for configuring the Location Broker and procedures for starting Location Broker processes.

After you set up the Location Broker services and build the binop_lu application, start the binop_lu server (after defining the foreign commands), specifying the address family as ip as shown in this example:

```
$ server ip
Registered: name'ip:elektra', port=1330
```

Your port number may differ from this one, because binop_lu uses dynamically assigned opaque ports.

After the server has registered itself, run the client, specifying the number of passes to execute as shown in this example:

```
$ client 4
pass   1; real/call: 20 ms
pass   2; real/call: 20 ms
pass   3; real/call: 10 ms
pass   4; real/call: 10 ms
```

Writing Interface Definitions 4

The first step in developing a distributed application is to define its interface or interfaces in Network Interface Definition Language (NIDL). A NIDL interface definition contains:

- A heading
- Import declarations
- Constant declarations
- Type declarations
- Operation declarations

The NIDL Compiler uses the information in an interface definition to generate header files and client and server stubs.

This chapter explains how to:

- Generate an interface Universal Unique Identifier (UUID) and a skeleton interface file
- Write an interface definition in NIDL
- Run the NIDL Compiler to produce the server and client stub files

This chapter shows the development of an interface definition for `binop_fw`, an application that uses the Location Broker forwarding facility to perform integer additions on a remote server. Chapter 5 describes how to develop and build the `binop_fw` client and server programs.

This chapter introduces NIDL through examples rather than syntax descriptions. For details of NIDL syntax, see Chapter 6.

4.1 Generating Interface UUIDs

Each object, type, and interface must have a UUID. You must generate a new UUID each time you create an object, type, or interface. You can create a UUID with the `uuid_gen` utility or in your application program with the `uuid_$gen` routine.

The `uuid_gen` utility is invoked as a foreign command, `uuid_gen`. Before invoking the command, define it as a foreign command in the RPC startup file, SYS$STARTUP:RPC$UCX_STARTUP.COM, as follows:

```
$ uuid_gen :== $rpc$exe:uuid_gen
```

After defining the foreign command, run the uuid_gen utility as shown in this example. The command generates an interface definition file in the C language syntax and places the output in the file binop_fw.idl.

```
$ define/user sys$output binop_fw.idl
$ uuid_gen -c
```

Note

This tool is available on VMS systems, ULTRIX systems, and other versions of the UNIX operating system. The command interface is common across all these systems, and therefore is not in a traditional DCL style.

For this command, precede qualifiers with a hyphen (-), rather than the customary slash (/).

You must define each DECrpc command as a foreign command.

Example 4-1 shows the interface file generated by uuid_gen.

Example 4-1: Interface File Generated by uuid_gen

```
%c [1]
[
uuid(41979f400000.0d.00.00.fb.40.00.00.00), [2]
version(1)
]
interface INTERFACENAME { [3]
}
```

[1] The first line of the skeletal definition is the syntax identifier, which is %c in this example, for the C language.

[2] The next part of the definition is the heading, which specifies a name, a UUID, and a version number for the interface.

[3] The last part of the definition is a pair of braces, between which go import, constant, type, and operation declarations. This chapter describes the syntax for the declarations.

By convention, the names of interface definition files end with the suffix .idl. To generate names for header, stub, and switch files, the NIDL Compiler replaces the suffix with .h, _cstub.c, _cswtch.c, and _sstub.c.

4.2 The Heading

The heading of an interface definition specifies the name and attributes of the interface.

4.2.1 Interface Names

After you have used `uuid_gen` to generate a skeletal interface definition, replace the dummy string "INTERFACENAME" with the name of your interface.

One naming convention uses interface names that end with an underscore, such as `rpc_` and `socket_`. Operation names begin with a dollar sign ($), so that operations in interfaces have names such as `rpc_$listen` and `socket_$equal`. Applications have interface names such as `bank` and `binop` and operation names such as `bank$deposit` and `binop$add`.

4.2.2 Interface Attributes

There are five interface attributes. Any interface that contains operations must specifiy at least the `uuid` attribute or the `local` attribute.

`uuid`
: The Universal Unique Identifier assigned to the interface by `uuid_gen`. No other object, type, or interface can be assigned this UUID.

`version`
: The version number of the interface. If you want several versions of an interface to coexist, you can distinguish them with version numbers.

`port`
: The well known port or ports on which servers exporting this interface will listen. In most cases, you should not use the `port` attribute; instead, you should allow the RPC run-time library to assign ports dynamically. See Chapter 1 for a discussion of well known ports.

`implicit_handle`
: The global variable containing handle information. If you do not specify this attribute, the handle must be passed as an explicit parameter to each operation.

`local`
: A flag that tells the NIDL Compiler to generate only header files (`.h`), not stubs. The interface definition should contain declarations only for constants and types, not for operations. If you specify the `local` attribute, the NIDL Compiler ignores any other interface attributes.

4.2.3 Examples of Interface Headings

The heading for the `binop_fw` interface definition specifies only an interface UUID, a version number, and the interface name:

```
[uuid(41979f400000.0d.00.00.fb.40.00.00.00), version(1)]
interface binop_fw
```

The heading for the `binop` application (see Chapter 3) specifies well known ports for the IP address family:

```
[uuid(41979f30a000.0d.00.00.fb.40.00.00.00),
    port(ip:[6677]), version(1)]
interface binop
```

4.3 Import Declarations

The NIDL `import` declaration is similar to the C `#include` directive. An `import` declaration specifies another interface definition whose types and constants are used by the importing interface.

The `import` declaration allows you to collect the declarations for types and constants that are used by several interfaces into one common file. For example, if you are defining two database interfaces named `lookup` and `update`, and these interfaces have many constants in common, you can declare those constants in a `db.idl` file and import this file in the `lookup.idl` and `update.idl` interface definitions:

```
[
uuid(41979f400000.0d.00.00.fb.40.00.00.00),
version(1)
]
interface lookup {

import 'db.idl';
}
```

Interface definitions can also use the `import` declaration to import one or more of the files supplied in the system `idl` directory, RPC$IDL. (You should never need to explicitly import `rpc.idl`, the interface definition for the RPC run-time library, since the NIDL Compiler automatically imports `rpc.idl` when it compiles any interface without the `local` interface attribute.)

The `-idir` qualifier of the NIDL Compiler allows you to specify a directory from which the Compiler will resolve the pathnames of imported files. You can thereby avoid putting absolute pathnames in your interface definitions.

Chapter 2 describes files in RPC$IDL.

4.4 Constant Declarations

The NIDL `const` declaration allows you to declare integer, character, or character string constants, as in the following examples:

```
[
uuid(41979f400000.0d.00.00.fb.40.00.00.00),
version(1)
]
interface music {

import 'music.idl';
const int array_size  100;
const char jsb  "Johann Sebastian Bach";
}
```

4.5 Type Declarations

NIDL provides a variety of data types, including simple types (such as integers, floating-point numbers, characters, and enumerations), constructed types (such as sets, strings, structures, unions, arrays, and pointers), and the `handle_t` type. The NIDL type declaration lets you give a name to any of these types.

The general form of a type declaration is

`typedef [` *type_attribute_list* `] type_specifier type_declarator_list;`

The *type_declarator_list* is optional.

This type declaration defines `integer32` as a name for a 32-bit integer type:

```
typedef   long integer32;
```

4.5.1 The Type Attributes handle and transmit_as

The type attributes `handle` and `transmit_as` specify characteristics of a named type.

The `handle` attribute specifies that a type can serve as a generic handle. You supply an autobinding routine to convert the generic handle type to the RPC handle type.

The `transmit_as` attribute associates a **transmitted type** that stubs pass over the network with a **presented type** that clients and servers manipulate. You supply routines that perform conversions between the presented and transmitted types.

One use of the `transmit_as` attribute is to help applications pass complex data types such as trees, linked lists, and records that contain pointers. The NIDL Compiler cannot generate code to marshall and unmarshall (copy data into and out of RPC packets) these data types, but the transmit_as attribute allows you to supply routines that convert the complex types into simpler types that can be marshalled and unmarshalled.

You can also use this feature to pass data more efficiently. For example, you might write routines that convert between sparse arrays and packed arrays; stubs transmit packed arrays over the network, and they present sparse arrays to the client and server programs. Chapter 7 illustrates this technique.

4.5.2 The Field Attributes last_is and max_is

The field attributes last_is and max_is can apply to members of structures and to parameters of operations. These attributes let you pass **open arrays** between clients and servers. An open array is an array whose length is determined at run time, when an operation that uses it is called. The last_is and max_is attributes control the amount of data transmitted between the client and server and the amount of storage allocated at the server.

The type declaration for a structure containing an open array must specify last_is and can also specify max_is. Chapter 7 includes a description of the last_is and max_is attributes and presents an example.

4.5.3 Examples of Type Declarations

The following example declares the type sockhandle_t as the textual representation of a socket address and specifies that this type is to be used as a generic handle:

```
typedef [handle] socket_$string_t sockhandle_t;
```

The interface definition for an example called sparse declares the type compress_t as a structure containing an open array, then declares two array types, compress_array and no_compress_array:

```
/* a run-length-encoded representation of an array */
typedef struct {
    int last;
    int [last_is(last)] data[CARRAY_SIZE];
} compress_t;

/* this type will be transmitted as a more compact type */
typedef [transmit_as(compress_t)] int compress_array[ARRAY_SIZE];

/* this type will be transmitted as is */
typedef int nocompress_array[ARRAY_SIZE];
```

For more examples of type declarations, look at the files in RPC$IDL, which contains interface definitions of structures used at run time, and in its c subdirectory, for C compiler include file formats. You can find representations of structures in these files so you will know the form if you want to extract information from a structure.

4.6 Operation Declarations

Operation declarations specify the signature of each operation in the interface, including the operation name, the type of data returned (if any), and the types of all parameters passed in the call. They also specify various field, parameter, and operation attributes.

The general form of an operation declaration is:

[*operation_attribute_list*] *type_specifier operation_declarator* (*parameter_list*) ;

The operation_attribute_list is optional. Each entry in the *parameter_list* specifies the type, attributes, and the name of a parameter.

This interface for a `sparse` operation contains the following declaration for the operation `sparse$compress_sum`:

```
[idempotent]
int sparse$compress_sum(
    handle_t [in] h,
    compress_array [in] array
    );
```

4.6.1 Operation Attributes

The operation attributes describe characteristics of an operation that affect communication between server and client. You can specify any of the following operation attributes:

- idempotent
- broadcast
- maybe
- comm_status

The `idempotent` attribute allows an operation to be executed more than once, not just once. This attribute allows the RPC run-time library to forego enforcement of the default ''at most once'' semantics. Specify `idempotent` for any operation that can safely be executed more than once. The `binopfw$add` operation is idempotent.

The `broadcast` attribute specifies that an operation be broadcast to all hosts on the local network, rather than delivered to a specific host. The RPC run-time library automatically applies idempotent semantics to any operation with the `broadcast` attribute. The use of this attribute is discouraged; see the discussion in Chapter 5.

The `maybe` attribute specifies that there is no need for confirmation that an operation has been executed. You can apply this attribute only if an operation has no output parameters and returns no value.

The `comm_status` attribute specifies that an operation returns a completion status. If a communications error occurs while the operation is executing, a cleanup handler in the client stub will catch the error and return the error code to the client.

4.6.2 Parameters

If an interface uses explicit handles, you must supply a handle as the first parameter in each operation declaration, as in the following example:

```
void exp$op(
    handle_t [in] h,
    int [in] a,
    int [in] b,
    int [out] c
    );
```

If an interface uses an implicit handle, you must specify the handle variable in an `implicit_handle` attribute of the interface, and the operations in the interface do not require handle parameters:

```
[uuid(338b5f985000.0d.00.00.37.27.00.00.00),
                implicit_handle(handle_t array_handle)]

void imp$op(
    int [in] a,
    int [in] b,
    int [out] c
    );
```

The `in` and `out` keywords in the preceding examples are parameter attributes. Section 4.2.2 describes the attributes you can apply to parameters.

4.6.3 Pointers as Parameters

NIDL pointers are really references: they must point to something and cannot be null.

In the C syntax of NIDL, you specify a pointer by preceding the parameter name with an asterisk (*). This construct is used primarily for output parameters, which, as in C, must be passed by reference. You can also use pointers to denote input parameters passed by reference.

The NIDL Compiler generates code that can marshall and unmarshall pointers only at top level and not within any constructed types. Chapter 7 describes the data type conversion mechanism that allows you to overcome this restriction.

4.6.4 Arrays as Parameters

In the C syntax of NIDL, you specify an array by placing the array length in brackets after the parameter name. Array subscripts start at 0. Arrays are always passed by reference, so an output array does not require a preceding asterisk. The following example specifies an array of 13 integers, indexed from 0 to 12, named `outputs`:

```
long [out] outputs[13]
```

NIDL also supports multidimensional arrays and open arrays. Chapter 6 explains array syntax in more detail.

4.6.5 Parameter Attributes

Characteristics of an operation parameter are specified by `parameter` attributes.

`in` The parameter is an input. It passes from client to server.

`out` The parameter is an output. It passes from server to client. In the C syntax of NIDL, an output parameter must be a pointer marked by the * operator.

`comm_status`
An operation returns a completion status. If a communications error occurs while the operation is executing, a cleanup handler in the client stub will catch the error and return the error code to the client.

4.6.6 The Field Attributes last_is and max_is

If you pass an open array (an array of variable length) as an operation parameter, use the `last_is` and `max_is` attributes to control how many elements are transmitted between the client and server and how much storage is allocated at the server. In operation declarations, field attributes appear together with parameter attributes, preceding the parameter. Chapter 6 includes descriptions of these attributes. Chapter 7 discusses the attributes in more detail and provides an example.

4.6.7 Examples of Operation Declarations

The `binop_fw` interface definition declares one operation, `binop_fw$add`:

```
[idempotent]
void binop_fw$add(
    handle_t [in] h,
    long [in] a,
    long [in] b,
    long [out] *c
    );
```

The next example shows one operation from among several in the `bank` interface definition. This operation declares the UUID as the RPC handle.

```
[ uuid(35c2c6a25000.0d.00.00.c3.66.00.00.00), version(1) ]
interface bank{

import 'nbase.idl';

type int bank$acct_t [32]
   .
   .
   .
void bank$inq_acct(
    uuid_$t       [in]  h,
    bank$acct_t [in]  acct,
    int           [out] balance,
    int           [out] trans_time,
    int           [out] create_time,
    );
   .
   .
   .
}
```

The interface definition for a `primes` procedure, declares a `primes$gen` operation:

```
[idempotent]
void primes$gen(
    handle_t      [in] h,
    int           [in, out] *last,
    int           [in] max,
    status_$t     [comm_status, out] *st,
    int           [in, out, last_is(last), max_is(max)] values[]
    );
```

4.7 The binop_fw Interface Definition

Example 4-2 shows the complete definition for the `binop_fw` interface.

Example 4-2: The binop_fw Interface Definition

```
%c
[uuid(4448ee491000.0d.00.00.fe.da.00.00.00), version(1)]

interface binopfw
{
[idempotent]
void binopfw$add(
    handle_t [in] h,
    long [in] a,
    long [in] b,
```

Example 4-2: (continued)

```
    long [out] *c
    );
}
```

4.8 Running the NIDL Compiler

After you have written the interface definition, run the NIDL Compiler to generate stub and header files. The syntax for the command is shown in this example:

`nidl` *filename* `[ -m | -s ] [`*other qualifiers*`]`

Note

This tool is available on VMS systems, ULTRIX systems, and other versions of the UNIX operating system. The command interface is common across all these systems, and therefore is not in a traditional DCL style.

For this command, precede qualifiers with a hyphen (-), rather than the customary slash (/).

You must define each DECrpc command as a foreign command.

The *filename* argument is the pathname of the interface definition file.

Specify either the `-m` qualifier or the `-s` qualifier. These qualifiers determine how stubs generated by the Compiler will dispatch remote procedure calls.

If you specify `-m`, the stubs will support multiple versions, multiple interfaces, or both within a single server, enabling you to build a server that exports more than one version of an interface. If you specify `-s`, the stubs will support only one version of an interface.

The NIDL compiler is invoked as the foreign command, `nidl`. Define the following foreign command in SYS$STARTUP:RPC$UCX_STARTUP.COM, the RPC start-up file:

```
$nidl :== $rpc$exe:nidl
```

After writing the interface definition and defining the command, run the `nidl` compiler as shown in this example:

```
$ nidl -s binop.idl -idir rpc$idl:
```

See the reference page for `nidl` in Chapter 16 for a complete description of the qualifiers.

The examples directory contains a `BUILD.COM` file that invokes the NIDL Compiler as follows:

```
$ nidl -s [.idl_d]bank.idl -idir rpc$idl:
```

The `-idir` option specifies a directory from which the compiler resolves pathnames of imported files.

On VMS systems, the compilation of `binop_fw.idl` generates files named `binop_fw.h`, `binop_fw_cstub.c`, `binop_fw_cswtch.c`, and `binop_fw_sstub.c`. These files are used to build the `binop_fw` client and server programs.

Developing Distributed Applications 5

After you have written interface definitions for a distributed application, you write a client program, write a server program, and build the application. This chapter expands upon the `binop_fw` application, whose interface definition was presented in Chapter 4.

5.1 The binop_fw Application

Table 5-1 compares the `binop_fw` example with the `binop` and `binop_lu` examples. In `binop_fw`, the user of the client program specifies a server host on the command line, and the server listens on an opaque port dynamically allocated by the RPC run-time library. The server registers with the Local Location Broker on its host so that the LLB can forward calls to the server port.

All three `binop` applications use explicit handles and manual binding. With manual binding the client code generates and binds an RPC handle that it passes as the first argument in its remote procedure calls.

Table 5-1: Comparison of the binop, binop_lu, and binop_fw Applications

Application	Server Host	Server Port	LB Registration	Call Delivery
`binop`	Specified on command line	Well-known	None	Direct to server port
`binop_lu`	Obtained from LB lookup	Opaque	Global and local	Direct to server port
`binop_fw`	Specified on command line	Opaque	Local only	From server host forwarding port

For applications in which the client knows where a server is running, use LLB forwarding, as illustrated in `binop_fw`. The server listens on an opaque port and does not require the server to register with the GLB. When the client makes its first remote procedure call, the server host LLB forwards the call to the server port. On return, the handle is fully bound, so that any subsequent calls go directly to the server port.

For applications in which the client does not know where a server is running, use Location Broker registration and lookup, which are illustrated in `binop_lu`. The server listens on an opaque port and registers its objects, interfaces, and socket address with the GLB. The client uses a Location Broker lookup call to obtain the server socket address and fully binds the handle to this address.

Have your applications use opaque ports with one of these two techniques rather than well known ports. (See the discussion of well known ports in Chapter 1.)

Complete source code for the `binop` example is in the examples directory. Chapter 3 includes descriptions of `binop` and `binop_lu`.

5.2 Data Types and Portability

When you develop distributed applications, the client and manager code that you write must conform to the interfaces that you define. The C data types used by your code must therefore be equivalent to the NIDL data types specified in your interface definitions.

Many systems (including most systems with Motorola MC680x0, Intel 80x86, Digital VAX, or IBM System/370 processors) support C scalar types that correspond straightforwardly and exactly to the NIDL scalar types. On other systems, however, C types that match the NIDL types may not exist. A NIDL type may also be matched by different C types on different systems.

The NIDL Compiler generates C code that uses data types defined by the Network Data Representation (NDR) protocol. Every NIDL scalar type maps to one NDR scalar type; this mapping is the same for all systems. The header file `RPC$INCLUDE:IDL_BASE.H` contains C definitions of the NDR types for particular systems. To ensure portability, you can use NDR data types to declare variables that correspond to scalars specified in your interface definitions. The examples in this manual often use the NDR types `ndr_$char`, `ndr_$short_int`, and `ndr_$long_int`.

5.3 Writing the Client

This section explains how to write a client program. Section 5-4 presents the `binop_fw` client code.

5.3.1 Client Structure

The source code for a client program consists of these elements:

- The header file generated from your interface definition by the NIDL Compiler

- The client application itself, that is, the user-written code that implements the client program and calls the remote procedures
- The client switch generated from the interface definition by the NIDL Compiler
- The client stub generated from the interface definition by the NIDL Compiler
- Any user-written code that performs autobinding or data type conversion (see Chapter 7)

If a client imports several interfaces, the client source code must include the header file, client switch, client stub, any autobinding routines, and any type conversion routines for *each* interface.

Table 5-2 lists the source files that make up the client in the `binop_fw` example. There are two application code modules: `client.c`, which contains the main program, and `util.c`, which contains utility routines that are used by both the client and the server.

Table 5-2: Client Source Code Files for the binop_fw Example

Source Code File	Module
`binop_fw.h`	Header file generated by the NIDL Compiler
`client.c`	Main program
`util.c`	Utility routines used by client and server
`binop_fw_cswtch.c`	Client switch generated by the NIDL Compiler
`binop_fw_sstub.c`	Client stub generated by the NIDL Compiler

5.3.2 Managing RPC Handles

When a client makes a remote procedure call, it must specify to the RPC run-time library the object that it is trying to access. The client uses an RPC handle to represent the object and the location of a server that can execute the call.

5.3.2.1 Binding Techniques – There are two binding techniques:

Manual binding	The client creates and manages RPC handles directly.
Automatic binding	The client uses generic handles instead of RPC handles. Whenever the client makes a remote procedure call, the stub calls a user-written **autobinding routine** that converts the generic handle into an RPC handle.

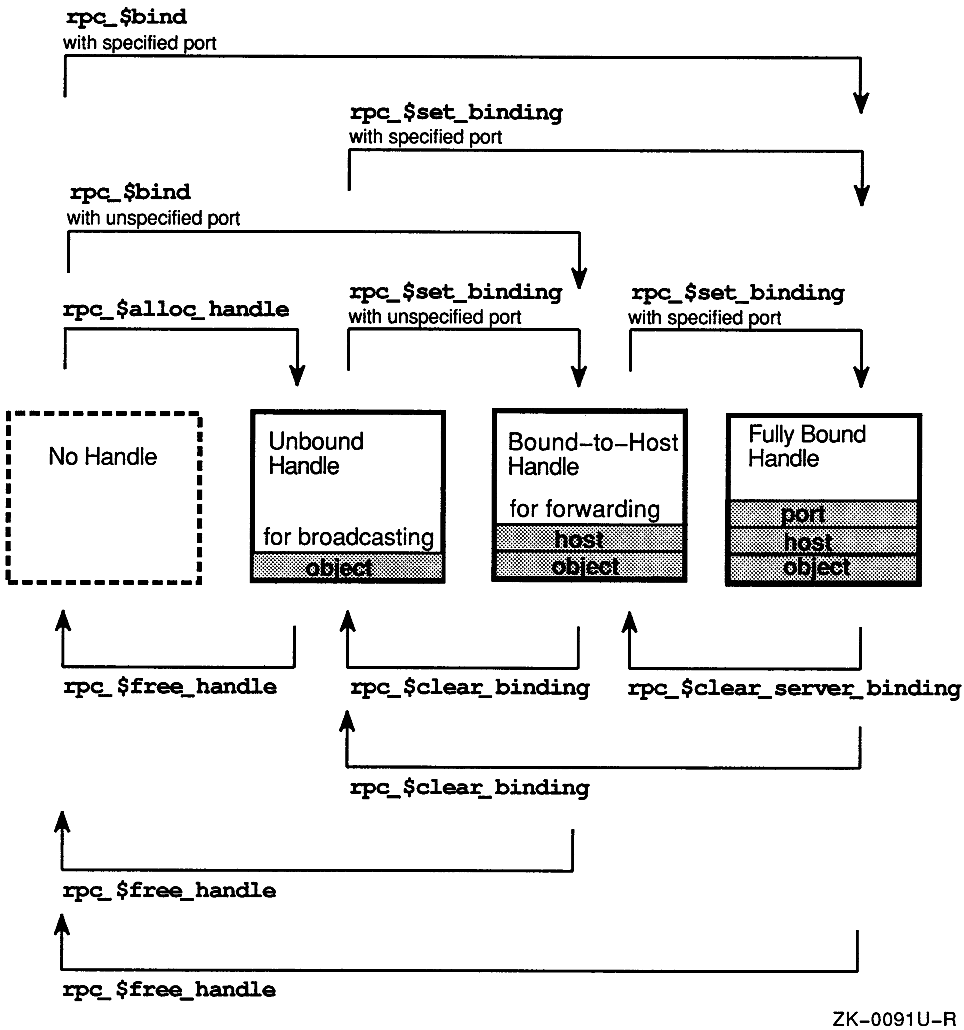
The binding technique determines where RPC handle management occurs, in client code or in autobinding code, but it does not affect how RPC handle management is implemented. You use the same library routines in both cases.

Like most of the examples in this book and in the online examples directory, `binop_fw` uses manual binding.

Chapter 1 discusses the differences between manual and automatic binding and compares the advantages and disadvantages of these techniques.

5.3.2.2 Overview of RPC Handle Management Routines – The RPC run-time library contains several routines that client applications can use to create handles, free handles, or change their binding states. Figure 5-1 illustrates the effects of these routines and shows the information represented in each possible binding state of an RPC handle. (See Section 5.3.4 for more information about RPC binding states.)

Figure 5-1: Calls That Manage RPC Handles and Their Binding States



5.3.2.3 Creating Handles – As Figure 5-1 illustrates, the `rpc_$bind` and `rpc_$alloc_handle` routines enable you to create an RPC handle in any binding state: fully bound, bound-to-host, or unbound.

The rpc_$bind routine takes as input an object UUID and a socket address. It creates a handle to represent the object and binds the handle to the socket address. You can create a fully bound handle by calling rpc_$bind with a fully specified socket address. You can create a bound-to-host handle by calling rpc_$bind with a socket address whose port number is socket_$unspec_port.

The rpc_$alloc_handle routine takes as input an object UUID. It creates an unbound handle to represent the object. You can use this handle to broadcast a remote procedure call, or you can invoke rpc_$set_binding to set its binding.

5.3.2.4 Changing Binding States

5.3.2.4 Changing Binding States – The rpc_$set_binding routine sets or resets the binding state in a handle. This routine enables a client to change the binding state without freeing and re-creating the handle. For example, if an application sequentially accesses several locations of an object, the client can:

1. Use rpc_$alloc_handle to create a handle.
2. Use rpc_$set_binding to bind to a server.
3. Make the remote procedure call to access the object.

Repeat steps 2 and 3, binding to servers on each host in sequence, to access all of the other objects.

The client does not need to call rpc_$clear_binding before it rebinds the handle to the next server, because rpc_$set_binding replaces any existing binding.

As with rpc_$bind, you can use rpc_$set_binding to obtain a bound-to-host handle, if you supply as input a socket address with a port number of socket_$unspec_port. You can use rpc_$clear_binding or rpc_$clear_server_binding to remove parts of the binding information in a handle.

5.3.3 Obtaining Socket Addresses

To obtain the socket address that rpc_$bind and rpc_$set_binding require as input, you can use a Location Broker lookup routine or the socket_$from_name routine.

5.3.3.1 Using Location Broker Lookup Calls – The Location Broker Client Agent offers routines that perform Location Broker lookups by object, type, interface, or any combination of these identifiers. Each lookup routine returns as output an array of database entries that match the specified criteria.

This chapter discusses the use of `lb_$lookup_interface`, which looks up servers by interface. The syntax and arguments for this routine are:

```
lb_$lookup_interface (&interface, &lookup_handle,
                      max_results, &num_results, results, &status);
```

The arguments are described here:

interface	an interface UUID
lookup_handle	a position in a Location Broker database
max_results	the maximum number of database entries that can be returned
num_results	the number actually returned
results	an array of the returned entries
status	the completion status

A client usually specifies `lb_$default_lookup_handle` as the value for `lookup_handle` in its first Location Broker lookup call; this value causes the lookup to start at the beginning of the database.

Chapter 3 described the `binop_lu` example, in which the client uses the Location Broker to find a server for the `binop_lu` interface. The client calls `lb_$lookup_interface` as follows:

```
status_$t st;
lb_$entry_t entry;
lb_$lookup_handle_t lookup_handle = lb_$default_lookup_handle;
unsigned long nresults;
. . .
do {
     lb_$lookup_interface(&binop_lu_v1$if_spec.id, &lookup_handle, 1L,
          &nresults, entry, &st);
     if (nresults < 1) {
          fprintf(stderr,
               "interface on valid family not found on lb_admin lookup\n");
          exit(1);
   }
} while (!socket_valid_family((long)entry.saddr.family, &st));
```

The `binop_lu` client initializes `lookup_handle` to the constant `lb_$default_lookup_handle`, which on input causes the lookup to begin at the start of the GLB database. The value 1L for `max_results` indicates that the routine can return at most one result; `nresults` is the number of entries that are actually returned.

If the lookup call returns an entry, the `binop_lu` client uses the routine `socket_$valid_family` to check that the address family for that entry is valid for the client host.

The max_results parameter specifies the maximum number of entries that a lookup routine can return (in the preceding example, one) and should not exceed the length of the results array.

If a lookup operation finds max_results entries before it has searched the entire database, it returns a value for lookup_handle that represents the start of the unsearched part of the database.

If a lookup operation reaches the end of the database before it finds max_results entries, it returns lb_$default_lookup_handle as the value of lookup_handle. Thus, a client can obtain all entries that match the lookup criteria by repeating the lookup call, using at each iteration the lookup_handle returned by the previous call, until the call returns lb_$default_lookup_handle.

Under normal conditions, repeated lookup calls obtain all matching entries in a database. However, some conditions can cause entries to be skipped or duplicated, for instance, if the database is modified between lookup calls. The client should be prepared to deal with missing or duplicated entries in the results array by retrying and verifying the answer or by using lb_$ routines or lb_admin to alter the database.

The routine may return an entry whose address families cannot be used by the host doing the lookup. The client program can protect against this by doing a global lb_$lookup_interface to get a list of all interfaces and verify that address families are valid. The client can also use the socket_$valid_families routine, which returns a list of the valid address families on the calling host.

Once the client has obtained the Location Broker entry for a server with a valid address family, it can use the socket address information in the entry to bind its handle. The binop_lu client calls rpc_$bind as follows:

```
h = rpc_$bind(&uuid_$nil, &entry.saddr, entry.saddr_len, &st);
if (st.all != status_$ok) {
    fprintf(stderr, "Can't bind - %s\n", error_text(st));
    exit(1);
}
```

The code uses the error_text routine, which is defined in util.c, to print any error message.

5.3.3.2 Converting Names to Addresses – If a client knows the name and the address family of the host it wishes to access, it can call socket_$from_name to obtain a socket address without using the Location Broker.

The socket_$from_name call requires a port number as one of its parameters. Unless the client knows the port number for a server, specify socket_$unspec_port. The run-time library will determine the port number at run time. The RPC run-time library extracts a port number, if one was specified in the NIDL definition of the interface, from the *interface*`$if_spec` variable. Otherwise, the port remains unknown, and the call is sent to the forwarding port at the host.

The `binop_fw` client, which knows the name of a host where a server is running but not a port number, uses `socket_$from_name` to convert the name into a socket address, then calls `rpc_$bind`:

```
socket_$from_name((long)socket_$unspec, (ndr_$char *) argv[1],
    (long) strlen(argv[1]), (long) socket_$unspec_port,
    &loc, &llen, &st);
h = rpc_$bind(&uuid_$nil, &loc, llen, &st);
```

5.3.4 Using RPC Binding States

The RPC run-time library has a different delivery mechanism for each of the three RPC binding states. This section describes how and why an RPC client might use fully bound, bound-to-host, and unbound handles.

5.3.4.1 Fully Bound Handles – When a client uses a fully bound handle to make a remote procedure call, the RPC run-time library sends the call directly to the host and port identified in the handle.

To obtain a fully bound handle, supply a fully specified socket address to either `rpc_$bind` or `rpc_$set_binding`. Any socket address obtained from a Location Broker will be fully specified. A socket address converted from a host name will not be.

Fully bound handles are always a direct and efficient means of communicating with a server.

5.3.4.2 Bound-to-Host Handles – When a program uses a bound-to-host handle to make a remote procedure call, the RPC run-time library sends the call to the host identified in the handle.

If a well known port was specified in the definition of the requested interface, the call is delivered to that port. Otherwise the call is delivered to the LLB forwarding port. The LLB forwards the call to the port on which the server is listening, provided a server for the requested object and interface has registered with it. When the call returns, the RPC run-time library at the client host then binds the handle to that port, and any subsequent calls are sent directly to the server.

You can obtain a bound-to-host handle in two ways:

- By calling `rpc_$bind` or `rpc_$set_binding` with an unspecified port in the socket address input parameter

- By calling `rpc_$clear_server_binding` on a fully bound handle

A client typically uses the first method (invoking `rpc_$bind` or `rpc_$set_binding`) after it uses `socket_$from_name` to generate a socket address. For example, the following code sends a matrix multiplication call to a server located at the host identified by `hostname`:

```
socket_$from_name (socket_$internet, hostname, hlen,
                   socket_$unspec_port, &saddr, slen, &st);
h = rpc_$bind (&matrix_id, &saddr, slen, &st);
matrix$multiply (h, a, b, result, &st);
```

A client typically uses the second method (invoking `rpc_$clear_server_binding`) after it has received an `rpc_$wrong_boot_time` error in `st.all`. If a client is fully bound to a server that exits and then restarts, listening on a new port, the client can reset the binding to the new port by calling `rpc_$clear_server_binding` on the existing handle; the handle will be rebound when the server responds to the next call.

Bound-to-host handles are most efficient when a client already knows the name or address of a host that is running the server it needs. For example, the client might be seeking a service that is provided by all hosts in the network, or the client might have been given the name of a particular host to access. The client does not need to do a Location Broker lookup. The server needs to register with the LLB on its host, but not with the GLB.

5.3.4.3 Unbound Handles – When a program uses an unbound handle to make a remote procedure call, the RPC run-time library broadcasts the call to all hosts on the local network. If a well known port was specified in the definition of the requested interface, the call is broadcast to that port. Otherwise, the call is broadcast to the LLB forwarding port.

You can obtain an unbound handle in two ways:

- By calling `rpc_$alloc_handle` to generate a new unbound handle
- By calling `rpc_$clear_binding` on an existing handle to clear the binding

You can also cause an operation to be broadcast by specifying the `broadcast` attribute in its NIDL declaration. If you make a remote procedure call to request an operation that has the `broadcast` attribute, the call is always broadcast, because the RPC run-time library automatically clears any binding of the handle before it issues the call. The client does not need to clear the binding before broadcasting again.

Instead of using unbound handles or specifying the `broadcast` attribute, it is preferable, whenever possible, to determine the address of a server host from a Location Broker lookup or the `socket_$from_name` routine. The broadcast delivery mechanism has several disadvantages:

- Not all systems and networks support broadcasting.
- Broadcasts are limited to hosts on the local network.
- Broadcasts make inefficient use of network bandwidth and processor cycles.
- The RPC run-time library does not support "at most once" semantics for broadcast operations; it applies idempotent semantics to all such operations.

All of these disadvantages pertain both to `broadcast` operations and to any operations that are called with unbound handles.

The RPC run-time library raises an error (`rpc_$comm_failure`, described in Section 5.3.8) if you attempt to make a call with an unbound handle, unless you have declared the operation to be idempotent.

The NIDL Compiler issues a warning if you specify the `broadcast` operation attribute without also specifying the `idempotent` attribute.

5.3.5 Identifying Servers

If a client application uses an unbound or bound-to-host handle to make a call, it may wish to identify the particular server that responded, for use in diagnostic or logging output. Because the handle is automatically bound to the responding server when the call returns, you can derive the location of the server from information in the returned handle.

The `rpc_$inq_binding` routine extracts a socket address from a handle. The `socket_$to_name` routine converts a socket address to a textual hostname. For example, a client might issue the following calls to report the location to which its handle is bound:

```
rpc_$inq_binding (h, &saddr, &slen, &st);
socket_$to_name (&saddr, slen, name, &namelen, &port, &st);
name[namelen] = 0;
printf ("bound to server on port %ld at host %s\n", port, name);
```

This technique works even for operations with the `broadcast` attribute. After a client receives a reply to a broadcast, the handle is fully bound, and the RPC run-time library does not clear the binding until the client uses that handle to issue another call.

5.3.6 Handling Errors

Distributed applications handle some errors in much the same way as local applications. For example, if a client issues a remote procedure call to request an operation, and the manager routine for the operation encounters a divide-by-zero error, that error is reflected to the client as if the server had been locally linked with the client.

However, a distributed application can also encounter errors that a purely local application would not. The next sections discuss the causes of three kinds of errors that are specific to remote procedure calls: communications errors, server failures, and interface mismatches.

5.3.6.1 Communications Errors – Communications errors occur in the underlying communications mechanisms, resulting in the failure of a client's request to reach the server or the failure of a server's response to reach the client. Communications errors are usually indicated by the `rpc_$comm_failure` status. To recover, a client can retry the failed call or try to find another server. Each reference page in Chapter 12 lists applicable RPC runtime library statuses.

You can use a status parameter, identified by the `comm_status` parameter attribute to check for communications errors. Chapter 4 describes status parameters.

5.3.6.2 Server Crashes – If a server crashes while handling a remote procedure call, an `rpc_$comm_failure` status is signaled to the client. To the client, the server failure is a form of communications error.

If the server fails and restarts between remote calls, the failure is usually indicated by an `rpc_$wrong_boot_time` status. A client can also receive an `rpc_$wrong_boot_time` status if one server fails and a different server starts, using the same port number as the failed server.

Recovery techniques depend on whether the client and the server maintain any state information between procedure calls:

- In a ''connectionless'' application, one that maintains no state between calls, the client needs only to rebind the handle. The client can call `rpc_$clear_server_binding`; then it can check whether the server has restarted. If the server did not restart, the client should unbind completely by calling `rpc_$clear_binding`, locate a new server, and rebind to the new server.
- In an application that does maintain some state between calls, the client must first clear the state (for example, by unwinding to the point at which it bound to the server), then rebind as in the connectionless case.

5.3.6.3 Interface Mismatches

5.3.6.3 Interface Mismatches – An interface mismatch occurs when the interface definition used to build a server differs from the interface definition used to build a client. If you increment the version number in the version interface attribute every time you change the interface definition, mismatches are easily detected and are indicated by an `rpc_$unk_if` status. If you do not increment the version number, the resulting errors may be difficult to diagnose.

In most cases, programs cannot recover from interface mismatch errors. To eliminate the errors, rebuild the out-of-date client or server.

If you want some clients to import an old version of an interface and some clients to import a new version, you can build one server that exports both versions of the interface. Chapter 7 describes how to build such a server.

You can add operations to an interface and maintain some backward compatibility without changing the version number, provided you do not change the signature or implementation of any existing operation. When you modify the interface definition, place declarations for new operations after all declarations for existing operations; that is, add new operations at the end of the interface, not in the middle.

Clients built with the old definition and servers built with the new definition will interoperate correctly. However, if a new client requests a new operation from an old server, the RPC run-time library will signal an `rpc_$op_rng_error` status. Example 5-1 shows how you can use a cleanup handler to check for an `rpc_$op_rng_error` status.

5.3.7 Using Cleanup Handlers

The RPC run-time library always signals a fault if an error occurs while it is handling a remote procedure call. Therefore, set cleanup handlers around remote procedure calls to catch and handle any such faults.

5.3.7.1 Initializing the Fault Management Routines – Before invoking any other DECrpc routines, a client or server should always invoke `pfm_$init` to initialize the fault management routines. This call causes C signals to be translated into signals that can be handled by the fault management routines. Attempts to use C signal handlers in the same program as fault management cleanup handlers can therefore result in unexpected behavior.

5.3.7.2 Setting and Releasing Cleanup Handlers – The `pfm_$cleanup` call sets a cleanup handler. The initial call to `pfm_$cleanup` returns as its value `pfm_$cleanup_set`, a status indicating that the cleanup handler is set; this call also returns as its output a cleanup record, a record of the context when the cleanup handler was set.

If a fault is signaled while a cleanup handler is set, these actions occur:

1. The process stack is unwound to the most recent `pfm_$cleanup` call.
2. The cleanup handler is released.
3. The `pfm_$cleanup` call returns the status value for the error that caused the fault.
4. Execution proceeds with the code that immediately follows the `pfm_$cleanup` call.

After you call `pfm_$cleanup`, test its return value, so that fault-handling code executes only if the value is an error status (indicating that an error has occurred), not if the value is `pfm_$cleanup_set` (indicating that the cleanup handler has just been set).

A cleanup handler typically ends either with code to continue back into the program or with a call to `pfm_$signal` or `pgm_$exit`. If the program will continue, it should call either `pfm_$reset_cleanup` or `pfm_$enable`.

The `pfm_$rls_cleanup` call releases a cleanup handler. Release a cleanup handler as soon as it is no longer necessary, so that fault-handling code is not executed inappropriately. For example, suppose a cleanup handler is set before a remote procedure call, and the cleanup handler contains code that prepares to retry the call. If you do not release the cleanup handler immediately after the call, a fault that occurs later in the program could cause the call to be executed again, unnecessarily.

In RPC applications, a cleanup handler is typically set just before a remote procedure call and released just after the call.

Section 5.3.6.3 explained how to create a new version of a server, which adds new operations to an interface, and maintain compatibility between existing clients, which call only the previously defined interface. and new servers, which export both old and new operations. The original server cannot execute the new operations for a client that needs to access the newer version of the interface. When such a client calls a new operation, it should be prepared to receive an `rpc_$op_rng_error` status.

Example 5-1 shows how a client might use a cleanup handler to check for `rpc_$op_rng_error` errors.

Example 5-1: Setting Up a Cleanup Handler

```
pfm_$cleanup_rec clrec;              /* set the cleanup handler */
st = pfm_$cleanup(&clrec);           /* test the return value */
/*
 * if an error occurred, clean up
 */
if (st.all != pfm_$cleanup_set) {
        if (st.all == rpc_$op_rng_error) {
        found an out-of-date server; find another one and rebind
        pfm_$reset_cleanup(&clrec, &st);
    }
    else {
         some other error occurred; report the error and exit
         pfm_$signal(st);
    }
}
/*
 * otherwise, proceed normally
 */
if$newop(h, input, &output);         /* call the operation */
pfm_$rls_cleanup(&clrec, &st);       /* release the cleanup handler */
```

5.3.7.3 Setting Multiple Cleanup Handlers – More than one cleanup handler can be in effect at once. If a program has set several cleanup handlers and a fault occurs, the most recently established cleanup handler is entered first, followed by the next most recently established cleanup handler, and so on to the first established cleanup handler if necessary.

5.3.7.4 Portability Considerations – The PFM package uses the C routines `setjmp` and `longjmp` to implement cleanup handlers. If you use local variables in fault-handling code, the unusual flow of control introduced by `setjmp` and `longjmp` can lead some optimizing C compilers to generate errant object code. You can circumvent this problem in a portable way.

If a local variable is modified after a cleanup handler is set but before the cleanup handler is invoked, the variable has an indeterminate value when referenced in the "fault-handling code path." To ensure that modifications made to the variable in the "normal code path" are visible to the fault-handling code, declare the variable with the ANSI C `volatile` qualifier.

Because `volatile` is not yet supported by all C compilers, the PFM header file defines a portable `Volatile` macro. This macro translates to `volatile` on systems whose compilers support the qualifier; on other systems it is null. It is recommended that programs that use local variables in cleanup handlers declare those variables `Volatile`. The code in Example 5-2 shows how to use a local variable portably in fault-handling code.

Example 5-2: Using Local Variables Portably in Fault Handling Code

```
Volatile boolean flag;
flag = false;
st = pfm_$cleanup(&crec);
if (st.all != pfm_$cleanup_set) {
    if (flag)
        release_pkt(pkt);
    pfm_$signal(st);
}
pkt = allocate_pkt();
flag = true;
```

more code
if a fault occurs here, the value of flag is indeterminate
more code

```
pfm_$rls_cleanup(&crec, &st);
```

Without the `Volatile` qualifier, the code in the example would not be portable. If a fault occurred at the point indicated, thereby invoking the cleanup handler, the value of `flag` would be indeterminate, and the cleanup handler would execute incorrectly.

5.3.8 Using the comm_status Parameter Attribute

The `comm_status` parameter attribute identifies a parameter as a `status parameter`. A status parameter provides a convenient way to check for communications errors in the execution of a remote procedure call. If you specify `comm_status` for an operation parameter, the NIDL Compiler puts a cleanup handler in the client stub routine for the operation. The cleanup handler catches any error with the `rpc_$mod` module code and passes the error to the client in the status parameter.

All `rpc_$` statuses have the `rpc_$mod` module code. The `rpc_$intro` page in Chapter 12 describes the `rpc_$` statuses.

5.3.8.1 Declaring Status Parameters in Interface Definitions – A status parameter must have the `comm_status` and `out` attributes and must be of type `status_$t`. The declaration of `primes$gen`, the operation in a `primes` application shown in Example 5-3, identifies a status parameter.

Example 5-3: Identifying a Status Parameter

```
[idempotent]
void primes$gen(
    handle_t [in] h,
    int [in, out] *last,
    int [in] max,
    status_$t [comm_status, out] *st,
```

Example 5-3: (continued)

```
    int [in, out, last_is(last), max_is(max)] values[]
    );
```

5.3.8.2 Checking Status Parameters in Client Programs – A client checks status parameters in the same way that it checks statuses returned by `rpc_$` calls or other RPC calls. The client in the `primes` example checks a status parameter after `primes$gen` returns, as shown in Example 5-4.

Example 5-4: Checking Status Parameters in Client Programs

```
primes$gen(h, &last, MAXVALS-1, &st, values);
/* check comm_status value */
if (st.all != status_$ok) {
    fprintf(stderr, "Error in rpc - %s\n", error_text(st));
    exit(1);
}
```

The `primes` client simply prints an error message and exits if the status parameter indicates an error. In other applications, the client might retry the call that failed or try to find another server, depending on the particular status that is returned.

5.3.8.3 Initializing Status Parameters in Manager Routines – If a remote procedure call executes without error, the value of its status parameter is not set. It is recommended, therefore, that the manager routine set the status parameter to `status_$ok` before it returns. Example 5-5 includes code from the `primes$gen` manager routine.

Example 5-5: Initializing Status Parameters in Manager Routines

```
void primes$gen(h, last, max, status, values)
handle_t h;
status_$t *status;
ndr_$long_int *last, max, values[];
{
    ndr_$long_int n, highest = values[0], index = 0;

    for (n = 2; n <= highest; n++)
        if (is_prime(n)) {
            values[index++] = n;
            if (index > max) break;
        }
    *last = index-1;
    status->all = status_$ok;
    return;
}
```

5.3.9 Using the comm_status Operation Attribute

NIDL also supports a `comm_status` operation attribute, which specifies that an operation returns a completion status. The client stub routine for such an operation contains a cleanup handler that catches any error with the `rpc_$mod` module code and returns the error code as its return value.

The manager routine for an operation with `comm_status` should be coded to return `status_$ok` if successful.

5.3.10 The binop_fw Client

The `binop_fw` client is the result of compiling four source code modules:

- `client.c`
- `util.c`
- `binop_fw_cstub.c`
- `binop_fw_swtch.c`

The switch and stub modules, of course, are generated by the NIDL Compiler from the interface definition. The `util.c` module contains a routine to print error messages; both the client and the server use this routine. The `main` routine is in the `client.c` module

5.3.10.1 The client.c Module – The client module contains directives to include three header files:

`binopfw.h`	The header file generated from the `binop_fw` interface definition
`socket.h`	The header file for the `socket_$` interface
`pfm.h`	The header file for the portable PFM interface

Example 5-7 shows the client module, `client.c`.

Example 5-7: The client.c Client Module for binop_fw

```
#include <stdio.h>
#include "binop_fw.h" 1
#include "socket.h"
#include <pfm.h>

#define CALLS_PER_PASS 100

globalref uuid_$t uuid_$nil;  2
extern long time();
extern char *error_text();

main(argc, argv)
int argc;
```

Example 5-7: (continued)

```
char *argv[];
{
    handle_t h;
    status_$t st;
    socket_$addr_t loc;
    unsigned long llen;
    socket_$string_t name;
    unsigned long namelen = sizeof(name);
    unsigned long port;
    ndr_$long_int i, n;
    int k, passes;
    int start_time, stop_time;

    if (argc != 3) {
        fprintf(stderr, "usage: client hostname passes\n"); [3]
        exit(1);
    }

    passes = atoi(argv[2]);

    pfm_$init((long) pfm_$init_signal_handlers);[4]

    socket_$from_name((long)socket_$unspec, (ndr_$char *) argv[1], [5]
        (long) strlen(argv[1]), (long) socket_$unspec_port,
        &loc, &llen, &st);
    if (st.all != status_$ok) {                                  [6]
        fprintf(stderr, "Can't convert name to sockaddr - %s\n",
            error_text(st));
        exit(1);
    }

   h = rpc_$bind(&uuid_$nil, &loc, llen, &st);  [7]
    if (st.all != status_$ok) {
        fprintf(stderr, "Can't bind - %s\n", error_text(st));

        exit(1);
    }

    rpc_$inq_binding(h, &loc, &llen, &st);  [8]
    if (st.all != status_$ok) {
        fprintf(stderr, "Can't inq binding - %s\n", error_text(st));
        exit(1);
    }

    socket_$to_name(&loc, llen, name, &namelen, &port, &st); [8]
    if (st.all != status_$ok) {
        fprintf(stderr, "Can't convert sockaddr to name - %s\n",
            error_text(st));
```

Example 5-7: (continued)

```
        exit(1);
    }

    name[namelen] = 0;

    printf("Bound to port %ld at host %s\n", port, name);

    for (k = 1; k <= passes; k++) {
        start_time = time(NULL);

        for (i = 1; i <= CALLS_PER_PASS; i++) {
            binop_fw$add(h, i, i, &n); [9]

            if (n != i+i)
                printf("Two times %ld is NOT %ld\n", i, n);
        }

        stop_time = time(NULL);

        printf("pass %3d; real/call: %2ld ms\n", [10]
            k, ((stop_time - start_time) * 1000) / CALLS_PER_PASS);
    }
}
```

[1] The client module contains directives to include `binop_fw.h`, the header file generated from the `binop_fw` interface definition, and `socket.h`, the header file for the `socket_$` interface. The handler file `binop_fw.h` contains an `include` directive for `rpc.h`, the header file for the `rpc_$` interface. The NIDL Compiler automatically puts such a directive in the header file it generates for any remote interface (that is, any interface without the `local` attribute).

[2] The module declares `uuid_$nil`, the nil UUID, as an external variable. The client uses `uuid_$nil` as the object UUID in its handle. The `globalref` declaration provides portability to VAX C. For other compilers, the `idl_base.h` header file, which is included by `rpc.h`, defines `globalref` as a synonym for `extern`.

[3] The client program takes two arguments: the network address of a host where a server is running and the number of passes to execute.

[4] After it has processed its arguments, the client calls `pfm_$init` to initialize the PFM package. This call should be made before calls to any other RPC routines.

[5] To convert the network address of the server host into a socket address, the client calls `socket_$from_name`, part of the socket address manipulation interface in the RPC run-time library. Because the port parameter for `socket_$from_name` is the predefined constant

socket_$unspec_port, the resulting socket address specifies a host, but not a particular port at that host.

6 After socket_$from_name returns, the client checks the completion status of the call, and if the status is not status_$ok, it prints an error message. Both the client and the server check the completion status of any call that returns a status. They use the error_text routine, which is defined in util.c, to print error messages.

7 The client supplies the address returned by socket_$from_name to rpc_$bind, which creates an RPC handle and binds this handle to the socket address. Because the address does not specify a port, rpc_$bind generates a bound-to-host handle. The object UUID in the rpc_$bind call is uuid_$nil, since binop_fw does not operate on any particular object.

8 For diagnostic and teaching purposes, the client in this example calls rpc_$inq_binding and socket_$to_name, so that it can print the host and port to which it is bound. Most real applications omit this step.

9 The first time the client calls binop_fw$add, the call is sent to the LLB forwarding port at the server host, and the LLB forwards the call to the server. On return, the handle is fully bound, so that all subsequent calls are sent directly to the server port.

10 After each pass, the client prints the real elapsed time per call.

After the last pass, the client exits.

5.3.10.2 The util.c Module – The util.c module in Example 5-8 contains only one routine, error_text. Both the client and the server use this routine to generate error messages.

Example 5-8: The util.c Module for binop_fw

```
#include "binop_fw.h"

char *error_text(st)
status_$t st;
{
    static char buff[200];
    extern char *error_$c_text();

    return (error_$c_text(st, buff, (sizeof) buff));
}
```

5.4 Writing the Server

This section explains how to write a server program.

5.4.1 Server Structure

The source code for a server program consists of the following elements:

- The header file generated from your interface definition by the NIDL Compiler
- The server initialization code, which registers the interface with the RPC run-time library and the Location Broker
- The manager code, which implements the operations in the interface
- The server stub generated from the interface definition by the NIDL Compiler
- Any user-written code that performs data type conversion

If a server exports several interfaces, the server source code must include the header file, manager code, server stub, and any type conversion routines for each interface.

Table 5-3 lists the source files that make up the server in the `binop_fw` example.

Table 5-3: Server Source Code Files for the binop_fw Example

Source Code File	Element
`binop_fw.h`	Header File generated from `binopfw.idl` by the NIDL Compiler
`server.c`	Main program, which contains server initialization code
`binop_fw.c`	Manager module
`binop_fw_sstub.c`	Server stub generated from `binopfw.idl` by the NIDL Compiler
`util.c`	Module containing utility routines used by both the client and the server

Manager procedures are independent of RPC routines and are exactly as they would be in a local implementation. The following subsections discuss server initialization code.

5.4.2 Writing Server Initialization Code

The server initialization code usually appears in the server main procedure (`main` in C). This code typically does the following:

- Processes any arguments supplied on the command line
- Creates the sockets on which it will listen
- Registers the server's objects and managers with the RPC run-time library
- Registers the server's objects and interfaces with the Location Broker
- Establishes termination and fault-handling conditions
- Begins listening for requests

The next sections describe each of these activities, using as an example the `binop_fw` server program, `server.c`.

5.4.2.1 Processing Arguments – The `binop_fw` server program performs several initialization tasks. It checks that there are the right number of input arguments; it checks that the specified address family is valid; and, just before it begins listening for requests, it prints a notification of its host and port.

The server takes as an argument the textual name of the address family `ip`. It calls `socket_$family_from_name` to convert this name into the integer representation that the `rpc_$` calls use, as shown in this example:

```
family = socket_$family_from_name((ndr_$char *) argv[1],
    (long) strlen(argv[1]), &st);
```

The server calls `socket_$valid_family` to check whether the specified address family is valid for the host on which it is running:

```
validfamily = socket_$valid_family (family, &st);
if (!validfamily) {
    printf ("Family %s is not valid\n", argv[1]);
    exit (1);
}
```

5.4.2.2 Creating Sockets – A single server can listen on several sockets at a time. However, a server that exports several interfaces can listen on one socket for requests for operations in any of those interfaces. Hence, most servers use only one socket per address family.

To obtain sockets on which to listen, a server calls `rpc_$use_family` or `rpc_$use_family_wk` once for each socket. The routine `rpc_$use_family` dynamically assigns an available opaque port, while `rpc_$use_family_wk` assigns the well known port that you specified in the interface definition. It is recommended that you avoid using well known ports, as discussed in Chapters 1 and 3.

The `binop_fw` server listens on one opaque port. It calls `rpc_$use_family` to obtain its socket:

```
rpc_$use_family (family, &loc, &llen, &st);
```

In this call, `family` is the integer representation of the address family specified on the command line, `loc` is the socket address for the port assigned by the RPC run-time library, and `llen` is the length of `loc`.

5.4.2.3 Registering with the RPC Runtime Library – As described in Chapter 3, a server can export several interfaces and can offer access through these interfaces to several types of objects. Each combination of interface and type requires a separate manager.

When the server RPC run-time library receives a remote procedure call from a client, it determines the correct manager to execute the call, based on the object and the operation requested, and dispatches the call to that manager. Every server must therefore inform the RPC run-time library about its managers and objects. A server calls `rpc_$register_mgr` once for each manager that it implements and calls `rpc_$register_object` once for each object that it supports.

The `binop_fw` server program makes the following call to register its manager with the RPC run-time library:

```
rpc_$register_mgr(
    &uuid_$nil,
    &binop_fw_v1$if_spec,
    binop_fw_v1$server_epv,
    (rpc_$mgr_epv_t) &binop_fw_v1$manager_epv, &st);
```

To register a manager, a server must supply a type identifier, an interface specifier, a server EPV, and a manager EPV. Because `binop_fw` does not involve any particular type, the `binop_fw` server specifies `uuid_$nil` as the type identifier. The interface specifier is defined in the header file, and the server EPV is defined in the server stub; both of these files are generated by the NIDL Compiler from your interface definition. You must define the manager EPV; typically this definition appears in the manager module.

Because `binop_fw` does not involve any particular object, the `binop_fw` server does not need to call `rpc_$register_object`.

5.4.2.4 Registering with the Location Broker – Most servers register their objects and interfaces with the Location Broker; clients can then use `lb_$` lookup calls to locate objects. A server must make a separate `lb_$register` call to register each possible combination of object, interface, and socket address. For example, the server should make six registration calls if it:

- Listens on one IP socket
- Exports two interfaces
- Manages three objects

Because the `binop_fw` application does not involve an object, its server specifies `uuid_$nil` as the object UUID for its Location Broker registration. Clients locate this server with Location Broker forwarding, so the server should register only with the Local Location Broker and not with the Global Location Broker.

The `binop_fw` server uses the following call to register with the Location Broker:

```
lb_$register (&uuid_$nil, &uuid_$nil, &binop_fw_v1$if_spec.id,
    (long)lb_$server_flag_local, (ndr_$char *) "binop_fw example",
    &loc, llen, &entry, &st);
```

This call specifies `uuid_$nil` for the object and type identifiers. The interface identifier is the `id` member of the `if_spec` for `binop_fw`, defined in the header file. To register only with the Local Location Broker, the server specifies `lb_$server_flag_local`. It supplies the text string ‘‘binop_fw example’’ as an annotation for the database entry. The `loc` specified in this call is the socket address that the server obtained from a call to `rpc_$use_family`.

5.4.2.5 Unregistering and Fault Handling – When a server starts, it should register itself with the RPC run-time library and with the Location Broker, so that clients can locate the server and communicate with it. When a server exits, it should unregister itself, so that clients do not continue trying to use it.

To unregister from the RPC run-time library, a server calls `rpc_$unregister`. In servers that export several interfaces or manage several objects, unregistrations should balance registrations: there should be an `rpc_$unregister` for every `rpc_$register_mgr` and an `lb_$unregister` for every `lb_$register`.

The code to unregister a server typically appears in a cleanup handler. The server sets the cleanup handler before it begins listening for requests. If the server receives a signal, it removes its registrations with the RPC run-time library and the Location Broker before exiting.

Following is the cleanup handler in the `binop_fw` server:

```
st = pfm_$cleanup(&crec);
if (st.all ! pfm_$clean_set) {
    status_$t stat;
```

```
fprintf(stderr, "Server received signal - %s\n",
    error_text(st));
lb_$unregister(&lb_entry, &stat);
rpc_$unregister(&binopfw_v1$if_spec, &stat);
pfm_$signal(st);
```

The code uses the `error_text` routine, which is defined in `util.c`, to print any error message.

5.4.2.6 Listening for Requests – To begin listening for requests, the server calls `rpc_$listen`. The first argument specifies the maximum number of requests that the server can process concurrently, in the DECrpc implementation, one (1).

The server uses this call to begin accepting requests from clients:

```
rpc_$listen ((long) 1, &st);
```

On normal completion, `rpc_$listen` does not return. However, the call will return on a catastrophic event or if an application issues a call to `rpc_$shutdown`. The shutdown call returns with `status_$ok`.

After a server creates sockets, registers objects and interfaces, and begins listening, it need not make any more calls. However, servers can register or unregister objects and interfaces while running, and they can also shut themselves down. A server can take these actions on its own or as part of its execution of client requests (in a manager routine).

5.4.3 Writing Manager Code

A manager implements the operations in one interface for objects of one type. In addition to defining a routine for each operation, the manager module defines the EPV through which these routines are called. Manager modules sometimes also require code to identify objects, to identify clients, or to register objects with the Location Broker.

5.4.3.1 Defining Manager EPVs – A manager EPV names the routines that implement the operations in an interface. The names of manager EPVs and manager routines are arbitrary, since these names appear only in code that you write, not in code that the NIDL Compiler generates. This manual uses a convention in which EPV names are similar to those of the client and server EPVs and routine names are similar to the operation names in the interface definition.

The `binop_fw` manager defines its EPV as follows:

```
globaldef binop_fw_v1$epv_t binopfw_v1$manager_epv {binop_fw$add};
```

Chapter 7 provides examples in which a server contains more than one manager or more than one version of a manager. In these examples, the manager EPVs help to distinguish different implementations of an interface.

5.4.3.2 Identifying Objects – In some applications, one manager supports several objects, and the manager must be able to identify the particular object on which the client wishes to operate. Clients in such applications typically use explicit handles, so that a handle passes from client to server with each call.

If the interface is manually bound, the manager can call `rpc_$inq_object` to extract the object UUID from the RPC handle. If, however, the interface is automatically bound, the handle must be either the object UUID itself or some other data type from which the manager can determine the UUID.

Example 5-9 shows a routine that checks to see if the object referred to by the RPC handle is the object expected. In the example, the `bankd` program passes the CheckObject routine a UUID, `h`. The routine compares the UUID to the known bank UUID.

Example 5-9: Checking the UUID in an Automatically Bound Interface

```
static boolean CheckObject(h, st)
uuid_$t *h;
status_$t *st;
{
    if (bcmp(h, &BankUUID, sizeof(BankUUID))) {
        fprintf(stderr, "(bankd) Request for wrong bank!\n");
        st->all = -1;        /* "object not found" */
        return(false);
    }

    st->all = status_$ok;
    return(true);
}
```

5.4.3.3 Identifying Clients – A server can identify clients from which it receives requests, for use in diagnostic or logging output. The RPC run-time library at a server host manipulates the location information in an RPC handle so that on the server side of an application, the handle specifies the location of the client making the call. Thus, just as a client can identify its server by extracting location information from a handle, a server can identify its client.

A manager routine might issue the following calls to report the location from which a server received a request:

```
rpc_$inq_binding(h, &loc, &llen, &st);
socket_$to_name(&loc, llen, name, &namelen, &port, &st);
```

```
name[namelen] = 0;
printf("Request from port %ld at host %s\n", port, name);
```

5.4.3.4 Registering Objects – In most applications, server initialization code registers the objects with the RPC run-time library and the Location Broker. However, if the server manages transient objects that it creates and deletes, have the manager routine that creates the objects register them, and have the manager routine that deletes objects unregister them.

5.4.3.5 Initializing Status Parameters – If an operation has a status parameter (a parameter with the `comm_status` attribute), have the manager routine that implements the operation set the status parameter to `status_$ok` before it returns.

5.4.4 The binop_fw Server

The `binop_fw` server is the result of compiling four source code modules: `server.c`, `binop_fw.c`, `util.c`, and `binop_fw_sstub.c`. The stub module is generated by the NIDL Compiler from the interface definition. Example 5-8 includes `util.c`, which contains a routine to print error messages. The manager module, `binop_fw.c`, contains the `binop_fw$add` routine that executes the actual addition operations. The `server.c` module performs all of the server initialization tasks.

5.4.4.1 The server.c Initialization Module – Example 5-10 contains the code for `server.c`.

Example 5-10: The server.c Module for binop_fw

```
#include <stdio.h>

#include "binop_fw.h"
#include "lb.h"              [1]
#include "socket.h"
#include <pfm.h>

globalref uuid_$t uuid_$nil;
globalref binop_fw_v1$epv_t binop_fw_v1$manager_epv;   [2]
extern char *error_text();

main(argc, argv)
int argc;
char *argv[];
{
    status_$t st;
    socket_$addr_t loc;
    unsigned long llen;
    unsigned long family;
```

Example 5-10: (continued)

```
boolean validfamily;
socket_$string_t name;
unsigned long namelen = sizeof(name);
unsigned long port;
lb_$entry_t entry;
pfm_$cleanup_rec crec;

if (argc != 2) {
    fprintf(stderr, "usage: serverfamily\n");
    exit(1);
}
pfm_$init((long)pfm_$init_signal_handlers); 3

family = socket_$family_from_name((ndr_$char *) argv[1], 4
    (long) strlen(argv[1]), &st);
if (st.all != status_$ok) {
    fprintf(stderr, "Can't get family from name - %s\n",
        error_text(st));
    exit(1);
}

validfamily = socket_$valid_family(family, &st); 5
if (st.all != status_$ok) {
    fprintf(stderr, "Can't check family - %s\n", error_text(st));
    exit(1);
}
if (!validfamily) {
    printf("Family %s is not valid\n", argv[1]);
    exit(1);
}

rpc_$use_family(family,&loc, &llen, &st); 6
if (st.all != status_$ok {
    fprintf(stderr, "Can't use family - %s\n", error_text(st));
    exit(1);
}

rpc_$register_mgr(    7
    &uuid_$nil,
    &binop_fw_v1$if_spec,
    binop_fw_v1$server_epv,
    (rpc_$mgr_epv_t) &binop_fw_v1$manager_epv,
    &st);
if (st.all != 0) {
    printf("Can't register manager - %s\n", error_text(st));
    exit(1);
```

Example 5-10: (continued)

```
    }

    lb_$register ( [8]
        &uuid_$nil,
        &uuid_$nil,
        &binop_fw_v1$if_spec.id,
        (long)lb_$server_flag_local,
        (ndr_$char *) "binop_fw example",
        &loc,
        llen,
        &lb_entry,
        &st);
    if (st.all != 0) {
        printf("Can't register - %s\n", error_text(st));
        exit(1);
        }

    socket_$to_name(&loc, llen, name, &namelen, &port, &st); [9]
    if (st.all != status_$ok) {
        fprintf(stderr, "Can't convert sockaddr to name - %s\n",
            error_text(st));
        exit(1);
    }

    name[namelen] = 0;
    printf("Registered: name'%s', port=%ld\n", name, port);

    st = pfm_$cleanup(&crec); [10]
    if (st.all != pfm_$cleanup_set) {
        status_$t stat;
        fprintf(stderr, "Server received signal - %s\n",
            error_text(st));
        lb_$unregister(&lb_entry, &stat);
        rpc_$unregister(&binopfw_v1$if_spec, &stat);
        pfm_$signal(st);
    }

    rpc_$listen((long) 1, &st); [11]
}
```

[1] The binopfw server module, like the client module, includes the binopfw.h, socket.h, and pfm.h header files. In addition, because the server makes Location Broker calls, the server module includes lb.h, the header file for the Location Broker Client Agent interface.

2 The server declares as an external variable the manager EPV `binopfw_v1$manager_epv`. The manager module defines this EPV. The server specifies the EPV when it registers its manager with the RPC run-time library.

3 Like the client, the server calls `pfm_$init` to initialize the PFM package before it makes any RPC calls.

4 The server program takes as an argument the textual name of an address family. It calls `socket_$family_from_name` to convert the textual name into the corresponding integer representation.

5 The call to `socket_$valid_family` checks whether the family is valid.

6 To obtain a socket on which to listen, the server supplies the address family, in its integer representation, to `rpc_$use_family`. The RPC run-time library assigns an available opaque port to the server; the run-time library returns the socket address for this port in the `loc` parameter.

7 To register its manager with the RPC run-time library, the server supplies the manager EPV to `rpc_$register_mgr`. The first parameter, the type UUID, is `uuid_$nil`, because the `binopfw` application does not involve any particular type.

8 To register with the Location Broker, the server calls `lb_$register`. It supplies the following information for its entry in the Location Broker database:

- An object UUID, in this case nil
- A type UUID, also nil
- An interface UUID, taken from the `if_spec`
- A flag that causes the entry to appear only in the Local Location Broker database
- An annotation
- A socket address

9 The server uses `socket_$to_name` to extract the host name and the port number from its socket address. It prints this information in a message.

10 Before it begins listening for requests, the server sets a cleanup handler. If the server receives a signal, it removes its registrations with the RPC run-time library and the Location Broker before exiting.

11 To begin listening for requests, the server calls `rpc_$listen`.

5.4.4.2 The binop_fw.c Manager Module – Example 5-11 contains code for the manager module.

The manager makes no RPC calls, so it includes only `binop_fw.h`, which defines `binop_fw_v1$epv_t` and declares the `binop_fw$add` operation.

Example 5-11: The binop_fw.c Manager Module for binop_fw

```
#include "binop_fw.h"

globaldef binop_fw_v1$epv_t binop_fw_v1$manager_epv = { binop_fw$add }; [1]
void binop_fw$add(h, a, b, c) [2]
handle_t h;
long a, b, *c;
{
    *c = a + b;
}
```

[1] The manager module defines `binopfw_v1$manager_epv`, the manager EPV. The `globaldef` provides portability to VAX C; for other C compilers, the `idl_base.h` header file in the `c` subdirectory of the system directory RPC$IDL defines `globaldef` as a macro with no replacement text.

[2] The manager module contains the implementation of the `binop_fw$add` procedure. The definition is just as it would be in a local application.

5.5 Building an Application

This section lists the usual steps in building a distributed application:

1. For each interface, run the NIDL Compiler to generate header files and to generate the source code for the server stub, the client stub, and the client switch.
2. For each interface, use the C compiler to generate object modules for the server stub, the client stub, and the client switch.
3. For each interface, compile any routines that perform automatic binding or data type conversion.
4. Compile the client application source to create the client object modules.
5. Compile the server initialization code and the managers to create the server object modules.
6. Link the client application object modules, the client switches, the client stubs, any automatic binding routines, and any type conversion routines to make the executable client.

7. Link the server and manager object modules, the server stubs, and any type conversion routines to make the executable server.

Remember that the client and the server must include the header files for any `lb_$`, `rpc_$`, `socket_$`, or `uuid_$` library routines or types they use; similarly, make sure that any interface definition that uses predeclared system types imports the corresponding NIDL file.

The NIDL files are located in the the RPC$IDL directory; the C header files are located in the `c` subdirectory.

The `SYS$SYSROOT:[SYSHLP.EXAMPLES]` directory includes a `README` file, a `BUILD.COM` file, and the source files for the `binop` client and server programs.

NIDL C Syntax 6

This chapter describes the C syntax of the Network Interface Definition Language (NIDL). This syntax of NIDL is a subset of ANSI C, with a few constructs added to express remote procedure call semantics.

Section 6.1 describes the overall structure of a NIDL interface definition. Sections 6.2 through 6.7 describe each of the elements in that structure. Section 6.8 is a detailed discussion of NIDL data types.

6.1 Interface Definition Structure

A NIDL interface definition file has the following structure:

```
%c
[ interface_attribute_list ] interface identifier
{
import_declarations
constant_declarations
type_declarations
operation_declarations
}
```

6.1.1 Syntax Identifier

The first line of an interface definition file identifies the syntax of NIDL in which the interface definitions are written. For the C syntax of NIDL, this identifier is `%c`.

6.1.2 Heading

The interface definition heading consists of three elements: an interface attribute list, enclosed in brackets; the keyword `interface`; and the interface identifier. Section 6.2 describes interface attributes in detail.

6.1.3 Body

The *interface definition body* follows the heading and consists of one or more of these declarations:

import_declaration Described in Section 6.3

constant_declaration	Described in Section 6.4
type_declaration	Described in Section 6.5
operation_declaration	Described in Section 6.6

There must be at least one constant, type, or operation declaration; a body containing only import declarations is not sufficient.

A semicolon terminates each declaration. Braces enclose the entire body.

6.1.4 Comments

As in C, `/*` and `*/` delimit comments as illustrated in this example:

```
/* all natural */
import 'cereal.idl';  /* no preservatives */
```

6.2 Interface Attributes

An interface definition heading specifies the name and attributes of the interface, as follows:

`[` *interface_attribute_list* `] interface` *identifier*

An *interface_attribute_list* is enclosed in brackets and includes one or more of the following elements, separated by commas:

`uuid (` *uuid_string* `)`
`version (` *version_number* `)`
`port (` *port_identifier_list* `)`
`implicit_handle (` *type_specifier identifier* `)`
`local`

If an interface definition contains any operation declarations, its heading must specify at least the `local` attribute or the `uuid` attribute.

6.2.1 UUID Attribute

The `uuid` attribute assigns a Universal Unique Identifier (UUID) to the interface. No other object, interface, or type can be assigned this UUID.

The `uuid` attribute has the following syntax, where *uuid_string* is the character-string representation of a UUID:

`uuid (` *uuid_string* `)`

6.2.2 Version Attribute

The `version` attribute helps you to manage multiple versions of an interface. It has the following syntax, where *version_number* is an integer:

`version (` *version_number* `)`

For example, if you were changing the parameters to a procedure in the `array` interface, the interface definition heading might look like this:

```
%c
[uuid(338b5f985000.0d.00.00.37.27.00.00.00), version (2)]
interface array
```

6.2.3 Port Attribute

The `port` attribute specifies the well known port or ports on which servers that export the interface will listen. In most cases, however, instead of using this attribute, allow the RPC run-time library to assign opaque ports dynamically. See Chapter 1 for a discussion of well known and opaque ports.

The `port` attribute has the following syntax:

`port (` *port_identifier_list* `)`

Entries in a *port_identifier_list* are separated by commas. Each entry has this form, where *family* is the address family and *port_number* is the well known port:

family`:` `[`*port_number*`]`

Specify at most one port per family. Table 6-1 lists the *family* values supported by NIDL.

Table 6-1: Family Values Supported by NIDL

Value	Address Family
`unspec`	Unspecified protocol
`unix`	Local to host (UNIX pipes, portals)
`ip`	Internetwork protocols (TCP, UDP)
`implink`	ARPANET Interface Message Processor (IMP) addresses
`pup`	XEROX PARC Universal Packet (PUP) protocols
`chaos`	MIT CHAOS protocols
`ns`	XEROX Network Systems (XNS) protocols
`nbs`	National Bureau of Standards (NBS) protocols
`ecma`	European Computer Manufacturers Association (ECMA)
`datakit`	Datakit protocols

Table 6-1: (continued)

Value	Address Family
`ccitt`	International Telegraph and Telephone Consultative Committee (CCITT) protocols (X.25, for example)
`sna`	IBM Systems Network Architecture (SNA) protocols
`unspec2`	Unspecified protocol

Although NIDL supports the families in the preceding list, the DECrpc run-time software supports only the IP address family. For example, the interface definition `binop.idl`, described in Chapter 3, specifies a well known port for the IP address family:

```
port(ip:[6677])
```

6.2.4 Implicit Handle Attribute

The `implicit_handle` attribute indicates that an interface uses implicit global variables rather than explicit operation parameters to represent objects.

The `implicit_handle` attribute has the following syntax:

`implicit_handle (` *type_specifier identifier* `)`

The *type_specifier* and *identifier* are the type and name of the global variable to be used as an implicit handle. The *type_specifier* must be either the RPC handle type `handle_t` or a generic handle type for which you have specified the `handle` type attribute.

If you specify an implicit handle for an interface, the client stub uses this handle to represent objects in all remote procedure calls and it passes no handle information to the server. Do not include handle parameters in the signatures of operations in the interface.

If you do not specify an `implicit_handle` in the interface definition heading, the interface uses explicit handles, and each operation must include a handle as the first parameter in its signature.

The interface definition heading for an interface that uses an implicit handle might look like this:

```
%c
[uuid(338b5f985000.0d.00.00.37.27.00.00.00),
                  implicit_handle(handle_t array_handle)]
interface array
```

Chapter 1 discusses handles and binding in detail.

6.2.5 Local Attribute

The `local` attribute indicates that the interface definition does not declare any remote operations; therefore, the NIDL Compiler generates only header files (`.h` files), not stubs.

If you specify the `local` attribute, the NIDL Compiler ignores any other interface attributes.

6.3 Import Declarations

The NIDL *import_declaration* is analogous to the C `#include` directive. It specifies an interface definition file that declares constants and types that the importing interface uses. It takes this form, where *file* is the pathname, enclosed in double quotation marks, of the file that you are importing:

import *file* ;

For example, the following declaration imports the definition for the `potato_` interface:

```
import "potato.idl";
```

The NIDL Compiler translates `import` declarations into C `#include` directives to include header files that correspond to the imported interfaces. However, if the imported interface contains operation declarations, the NIDL Compiler does not generate stub procedures for these operations. For example, if the interface definition `foo.idl` contains an import declaration for the `potato_` interface, then the NIDL Compiler will generate a C header file named `foo.h` that contains the following `#include` directive:

```
#include "potato.h";
```

The stub files that the Compiler generates, however, will contain no procedures for the `potato_` operations.

Importing an interface many times has the same effect as importing it once.

6.4 Constant Declarations

The NIDL *constant_declaration* takes the form

const *type_specifier identifier integer* | *string* | *value_identifier* ;

The *type_specifier* is the data type of the constant you are declaring, *identifier* is the name of the constant, and *integer*, *string*, or *value_identifier* is the value you are assigning to the constant. A *value_identifier* can be any previously defined constant.

The C syntax of NIDL provides only `int` and `char` constants. NIDL does not support constant expressions. The following are examples of constant declarations:

```
const int MAX = 100;
const CHAR DSCH = "Dmitri Shostakovich";
```

6.5 Type Declarations

The NIDL *type_declaration* lets you give a name to a data type. It takes the following form:

`typedef [` *type_attribute_list* `]` *type_specifier type_declarator_list ;*

The *type_attribute_list* is optional.

Some of the constructs that appear in type declarations can also appear in the parameter lists of operation declarations. Section 6.6 describes the use of these constructs in operation declarations. Section 6.7 describes NIDL data types in detail.

6.5.1 Type Attributes

The optional *type_attribute_list* includes one or both of the following elements, separated by commas:

`handle`
`transmit_as (` *xmit_type* `)`

These attributes can appear only in `typedef` declarations.

6.5.1.1 The handle Attribute – The `handle` attribute specifies that a type can serve as a generic handle. You must supply automatic binding routines to convert this type to `handle_t`, the RPC handle type.

The following example declares a generic handle type, `filehandle_t`, which is a structure containing the textual representations of a host and a pathname:

```
typedef [handle] struct {
    socket_$string_t host;
    char path[1024];
    } filehandle_t;
```

Chapter 7 discusses automatic binding and autobinding and autounbinding routines, and describes an application that uses UUIDs as generic handles.

6.5.1.2 The transmit_as Attribute – The `transmit_as` attribute associates a **transmitted type** that stubs pass over the network with a **presented type** that clients and servers manipulate. You must supply routines that perform conversions between the presented and transmitted types.

There are two primary uses for this attribute:

- To pass complex data types for which the NIDL Compiler cannot generate marshalling and unmarshalling code. Such types include trees, linked lists and structures that contain pointers.
- To pass data more efficiently. An application can provide routines to convert a data type between a sparse representation (presented to the client and server programs) and a compact one (transmitted over the network).

The *xmit_type* in a `transmit_as` attribute must be a named type defined previously in another type declaration; it indicates the transmitted type that the stubs will pass between client and server.

The following `typedef` statements declare presented and transmitted types for a linked list:

```
typedef struct {
    int last;
    int [last_is(last)] values[MAXELEMENTS];
    } trans_t;
typedef [transmit_as(trans_t)] struct {
    int value;
    list_t *next;
    } list_t;
```

Because `list_t` contains a pointer to a `list_t`, the NIDL Compiler cannot generate code to marshall this data type. Instead, it generates code that calls user-written routines to convert between `list_t` and `trans_t`, and the stubs transmit the linked lists as `trans_t` structures.

Chapter 7 discusses type conversion, specifies the signatures for conversion routines, and describes two applications that use type conversion.

6.5.2 Type Specifiers

The *type_specifier* portion of a *type_declaration* can specify any of the following:

Simple types

`int`	`unsigned`	`float`	`byte`
`long`	`unsigned long`	`double`	`void`
`short`	`unsigned short`	`char`	`enum`
`small`	`unsigned small`	`boolean`	`short enum`

Constructed types

`bitset`	`union`
`string0`	arrays
`struct`	pointers

The RPC handle type `handle_t`

Named types defined with `typedef` declarations

Section 6.7 describes these types in detail.

6.5.3 Field Attributes

NIDL provides two field attributes that apply only to arrays: `last_is` and `max_is`. These attributes identify *last* and *max* fields that at run time will supply the stubs with information about the length of an array; `last_is` and `max_is` are typically used for an open array, an array whose declaration does not specify an explicit fixed length.

An array with `last_is` or `max_is` must be either a member of a structure or a parameter of an operation. These attributes therefore can appear either in type declarations or in operation declarations. The attributes precede the array name in a *field_attribute_list*:

type_specifier [*field_attribute_list*] *array_declarator* [*array_length*]

The *field_attribute_list* comprises one or both of the following elements, separated by commas:

`last_is` (*last*)
`max_is` (*max*)

The `last_is` attribute identifies another field, *last*, that at run time will be the index of the last array element to be passed. Client and server programs use this field to dynamically indicate the size of an array.

The `max_is` attribute identifies another field, *max*, that at run time will be the maximum possible index of the array. Client programs use this field to dynamically indicate the maximum size of an array.

The following type declaration defines a structure that contains an open array, its *max*, and its *last*:

```
typedef struct {
    int pmax;
    int plast;
    int [max is(pmax), last_is(plast)] parray[];
    } pixels;
```

See Chapter 7 for a detailed discussion of `last_is` and `max_is`.

6.5.4 Type Declarators

The *type_declarator_list* specifies names for a particular type. To include more than one name in a list, separate the names with commas. For example:

```
typedef long integer32, int32;
```

6.5.4.1 Pointers – To specify a pointer type, precede the name with an asterisk. For example:

```
typedef int *pointer_to_int;
```

6.5.4.2 Arrays – To specify an array type, put brackets after the name. Inside the brackets you can supply the array size, an asterisk, or nothing. If you supply an asterisk or you supply nothing, you are declaring an open array (one whose length will not be known until run time), and you must apply the `last_is` field attribute to the array. Array subscripts start at 0. The following example of a `struct` includes two arrays:

```
typedef struct {
    char    fixed[32];
    int     last;
    char    [last_is (last)] open[];
    } arrays;
```

In a `struct` that contains an open array, the array must be the last member. A `union` cannot contain an open array. See Chapter 7 for more information about open arrays.

Use consecutive pairs of brackets to declare multidimensional arrays, as in C. For example:

```
typedef int two_by_four [2][4];
```

Only the first dimension of a multidimensional array can be unspecified. For example:

```
typedef int n_by_four [][4];    /* this is valid */
typedef int two_by_n [2][];   /* this is NOT valid */
```

6.6 Operation Declarations

The NIDL *operation_declaration* is analogous to a C function heading. An operation declaration has the following form:

[*operation_attribute_list*] *o_type_specifier operation_declarator* (*parameter_list*) ;

Entries in a *parameter_list* are separated by commas. Each entry has the following form:

p_type_specifier [*field_attribute_list parameter _attribute_list*] *parameter_declarator*

The following subsections discuss the parts of an operation declaration.

6.6.1 Operation Attributes

The optional *operation_attribute_list* includes one or more of the following keywords, separated by commas:

```
idempotent
broadcast
maybe
comm_status
```

6.6.1.1 The idempotent Attribute – By default, the RPC run-time library provides ‘‘at most once’’ call semantics. These semantics ensure that an operation, when called once, is executed not more than once. They require the server to save the results of an operation until the client acknowledges its receipt of those results.

The `idempotent` attribute specifies that an operation can be executed any number of times. If an operation is idempotent, the server does not need to save results and the client does not need to issue acknowledgements, so performance is improved. Use the `idempotent` attribute for any operation that can safely be executed more than once; for instance, an operation that simply reads a value is idempotent, while one that increments a value is not.

6.6.1.2 The broadcast Attribute – The `broadcast` attribute specifies that the RPC run-time software should always broadcast an operation to all hosts on the local network. The broadcast is to a well known port if one has been specified and to the Local Location Broker forwarding port if it has not. When a client calls an operation with the `broadcast` attribute, the run-time software automatically clears any binding from the handle before issuing the remote procedure call.

The RPC run-time library applies idempotent call semantics for all broadcast operations, so it executes any operation with the `broadcast` attribute as though the operation also had the `idempotent` attribute. For clarity, explicitly specify `idempotent` whenever you specify `broadcast`; if you do not, the NIDL Compiler issues a warning.

Because of the disadvantages listed in Chapter 5, avoid using the `broadcast` attribute. See the discussion of unbound handles and broadcasting in Chapter 5.

6.6.1.3 The maybe Attribute – The `maybe` attribute specifies that the caller of an operation does not expect any response and that the RPC run-time software need not guarantee delivery of the call. Operations with this attribute cannot have any output parameters and cannot return anything. You might use `maybe` for an operation that posts a notification whose receipt is not crucial.

6.6.1.4 The comm_status Attribute – The `comm_status` attribute specifies that an operation returns a completion status, a status code of type `status_$t`. If a communications error occurs while the operation is executing, a cleanup handler in the client stub will handle the error and return the error code as the return value of the operation. Code the manager routine for an operation with the `comm_status` attribute to return `status_$ok` if successful.

NIDL also supports a `comm_status` parameter attribute; this attribute identifies an output parameter that will reflect status and hence provides functionality similar to that of the `comm_status` operation attribute. Chapter 5 describes the use of status parameters.

6.6.2 Operation Type Specifiers

The *o_type_specifier* is the data type that the operation returns. It can be any scalar type or previously named type, but it cannot be a pointer. For example, if the operation returns a short integer, specify `short` as the *o_type_specifier*. Specify `status_$t` if the operation has the `comm_status` operation attribute. Specify `void` if the operation does not return. If you omit the *o_type_specifier*, the operation must return an `int`.

6.6.3 Operation Declarators

The *operation_declarator* is the name of the operation.

6.6.4 Parameter Lists

The parameters of an operation appear in a *parameter_list*. The entry for each parameter takes the following form:

p_type_specifier [*field_attribute_list parameter_attribute_list*] *parameter_declarator*

Use commas to separate the entries in a *parameter_list*.

If an interface uses explicit handles, the first parameter in the *parameter_list* for each operation must be the explicit handle. If an operation uses manual binding, the handle must have the type `handle_t`.

6.6.4.1 Parameter Type Specifiers – The *p_type_specifier* specifies the data type of the parameter.

6.6.4.2 Field Attributes and Parameter Attributes – The *field_attribute_list* can include `last_is` and `max_is` and can apply only to array parameters. The associated *last* and *max* must also be parameters in the *parameter_list*. Section 6.5.3 describes field attributes; Chapter 7 discusses them in further detail and presents an example. The *parameter_attribute_list* can include the following attributes:

`in`	The parameter is an input. It passes from client to server, that is, from the calling routine to the called routine.
`out`	The parameter is an output. It passes from server to client, that is, from the called routine to the calling routine. Output parameters are passed by reference and must be either pointers or arrays.
`comm_status`	The parameter is a status parameter. If a communications error occurs, a cleanup handler in the client stub will handle the error and pass the error code to the client in this parameter.

Every parameter must have at least one of the directional attributes `in` and `out`. A list including both `in` and `out` indicates that the parameter passes in both directions.

A parameter with the `comm_status` attribute must be of type `status_$t` and must also have at least the `out` attribute. Chapter 5 describes the use of status parameters.

Field attributes and parameter attributes can appear in any order. If a parameter has more than one attribute, separate the attributes with commas.

6.6.4.3 Parameter Declarators – The *parameter_declarator* specifies the name of each parameter. By default, `in` parameters are passed by value. To denote an `in` parameter that is passed by reference, precede the *parameter_declarator* with an asterisk (`*`). This construct is typically used when the application software is implemented in Pascal.

All `out` parameters are passed by reference. Unless the parameter is an array, you must precede the *parameter_declarator* with an asterisk (`*`).

Use brackets to specify arrays. The syntax for array parameters is the same as for array types, described in Section 6.5.4.

6.6.5 Examples

The following example declares an operation named `simple$op` that takes no parameters, returns no value, and need not be executed:

```
[maybe] void simple$op();
```

The interface definition for an `xmitas` application declares the `xmitas$sum` operation. This idempotent operation returns an integer. Its input parameters are an explicit RPC handle and a list structure of the named type `list_t`:

```
[idempotent]
int xmitas$sum(
    handle_t    [in] h,
    list_t      [in] list
    );
```

The interface definition for a `primes` application declares the `primes$gen` operation. This operation does not return a value. Its parameters include two pointers and an open array. Its declaration illustrates the use of operation attributes, field attributes, and parameter attributes:

```
[idempotent]
void primes$gen(
    handle_t    [in] h,
    int         [in, out] *last,
    int         [in] max,
    status_$t   [comm_status, out] *st,
    int         [in, out, last_is(last), max_is(max)] values[]
    );
```

6.7 Data Types

This section describes in detail the *type_specifier* expressions that you can use in type declarations and in the parameter lists of operation declarations. These expressions can specify simple types, constructed types, named types, or the RPC handle type `handle_t`.

6.7.1 Simple Types

NIDL supports a variety of simple data types including integers, floating-point numbers, characters, `boolean`, `byte`, `void`, and enumerations:

Integer Types

Type	Size
int	32 bits
long	32 bits
short	16 bits
small	8 bits
unsigned	32 bits
unsigned long	32 bits
unsigned short	16 bits
unsigned small	8 bits

You can include the keyword `int` after any of the other integer type names. For example, `long` and `long int` are synonymous.

Floating-Point types

Type	Size
float	32 bits
double	64 bits

The `byte` Type

The integer types listed in the integer type table are subject to data conversion when the native data representation formats of client and server hosts differ. The `byte` type is an 8-bit integer whose representation format is guaranteed not to be converted. You can protect data of any type from data conversion by transmitting that type as an array of `byte`; Chapter 7 discusses the use of transmitted types.

The `char` Type

The character type, `char`, is unsigned. NIDL does not support a signed character.

The `boolean` Type

Following C convention, a value of 0 means ''false,'' and any nonzero value means ''true.''

The `void` Type

This type is used for an operation that does not return a value.

Enumerations

```
enum { identifier_list }
short enum { identifier_list }
```

The enumerated types provide names for integers. An `enum` is a 32-bit integer; a `short enum` is a 16-bit integer. You can declare these types only in `typedef` statements. The NIDL Compiler assigns integer values, beginning at 0, to `enum` identifiers based on their order in *identifier_list*. For example:

```
typedef enum {John, Paul, George, Ringo} beatles;
```

In this declaration, `John` gets the value 0, `Paul` gets 1, `George` gets 2, and `Ringo` gets 3.

6.7.2 Constructed Types

NIDL also supports constructed data types, including sets, strings, structures, discriminated unions, pointers, and arrays:

Sets

```
bitset enum { identifier_list }
short bitset enum { identifier_list }
```

A `bitset` is similar to an enumeration, but instead of defining names for integers, it defines names for bits in a single 32-bit integer, starting with the least significant bit. A `short bitset` defines names for bits in a 16-bit integer. For example:

```
typedef bitset enum {Steinhardt, Dalley, Tree, Soyer} guarneri;
```

In this declaration, `Steinhardt` represents the value of bit 0 in an integer, `Dalley` represents bit 1, `Tree` represents bit 2, and `Soyer` represents bit 3.

Strings

```
string0 [ length ]
```

A `string0` is a C-style null-terminated string, that is, a character array whose last element is the null character `\0`. The *length* indicates the maximum length of the string, including the terminating zero byte. For example:

```
string0[7]
```

The specified string is long enough to hold ''Ligeti''.

Structures

```
struct tag {
      type_specifier [ field_attribute_list ] declarator ;
      ...
      }
```

A NIDL `struct` cannot contain pointers unless you apply the `transmit_as` type attribute and supply routines to convert the structure to a transmissible type. The *tag* is optional.

The *field_attribute_list* can apply only to arrays. Section 6.5.3 describes field attributes.

An open array can appear in a structure only as the last member. A structure containing an open array must be passed by reference.

Unions

```
union switch ( d_type_specifier discriminator ) tag {
      case constant : type_specifier declarator ;
      ...
      default : type_specifier declarator ;
      }
```

A NIDL `union` must be discriminated and hence differs considerably from its C counterpart. In the union header, you specify a discriminator and its type; the discriminator selects a member at the time the union is used. The NIDL `union` is a combination of C `union` and `switch` syntax.

The *d_type_specifier* and the *discriminator* are the type and the name of the discriminator. The *d_type_specifier* must be one of the simple types described in Section 6.7.1. The NIDL Compiler uses the optional *tag* to generate identifiers in source code representations of the union; see Section 6.7.5.

A default member, identified by the label `default`, can optionally appear anywhere in the list of cases. At the time the union is used, if the value of *discriminator* does not match any *constant* in the list of cases, the default member applies. In the absence of a default member, failure to match a *discriminator* raises an error.

The NIDL Compiler can generate C source code to represent a union with a `default` case.

To indicate that several cases take the same declarator, omit the *type_specifier*, the *declarator*, and the semicolon in all but the last `case`. To indicate an empty member, omit the *type_specifier* and the *declarator*. For example:

```
typedef union switch ( int pick ) {
            case 1 :
            case 2 : int fraise;
            case 3 : float framboise;
            case 4 :
            case 5 : ;
            } berries;
```

A `union`, like a `struct`, cannot contain pointers unless you apply the `transmit_as` type attribute and supply routines to convert the union to a transmissible type.

Section 6.7.5 discusses how the NIDL Compiler represents discriminated unions in the C source code it generates.

Pointers

*type_specifier *identifier*

To specify a pointer, precede the identifier with an asterisk (*). For example:

```
int *pointer_to_int
```

A NIDL pointer cannot be null.

The NIDL Compiler generates code that can marshall and unmarshall pointers only ''at top level'' and not within any constructed types. You can overcome this restriction by applying the `transmit_as` type attribute and supplying routines to convert the constructed type to a transmissible one.

Arrays

type_specifier identifier [*length*]

To specify an array, follow the name with brackets enclosing the number of elements in the array. If *length* is an asterisk or is omitted, the array is open. Consecutive pairs of brackets specify a multidimensional array. Section 6.5.4 describes array syntax in more detail.

6.7.3 The RPC Handle Type

The `handle_t` type denotes an opaque handle type meaningful to the RPC run-time library. If you specify this type for the explicit handles or the implicit handle in an interface, the interface uses manual binding.

6.7.4 Named Types

Named types are types defined by type declarations. For example, the following `typedef` statement defines `long_int` to be a synonym for `int`:

```
typedef int long_int;
```

Section 6.5 describes type declarations in detail.

6.7.5 Representation of Unions

NIDL unions are discriminated, unlike C unions. When the NIDL Compiler generates C code to represent a NIDL union, it embeds the union and the discriminator in a C structure. The name of the NIDL union becomes the name of the C structure. If you assign a tag to the NIDL union in your type declaration, the compiler uses the tag to name the embedded C union; otherwise, the compiler uses a generic name.

The following declaration assigns `utag` as the tag for a union named `union_with_tag`:

```
typedef union switch (short i) utag {
    case 1:
    case 2:
        struct { short a, b; } struct1;
    case 3:
    case 4:
        struct { float x, y; } struct2;
    case 5:
        char p;
    case 6:
        char q;
    } union_with_tag;
```

In the C definition that the NIDL Compiler generates, the union name `union_with_tag` becomes the name of the embedding structure, and the tag `utag` becomes the name of the embedded union:

This example of NIDL Compiler output shows code reformatted for readability and with comments added:

```
typedef struct union_with_tag union_with_tag;
struct union_with_tag {
    ndr_$short_int i;            /* the discriminator */
    union {                      /* the union */
        /* case(s): 1, 2 */
        struct {
            ndr_$short_int a;
            ndr_$short_int b;
            } struct1;
        /* case(s): 3, 4 */
        struct {
            ndr_$short_float x;
            ndr_$short_float y;
            } struct2;
        /* case(s): 5 */
        ndr_$char p;
        /* case(s): 6 */
        ndr_$char q;
        } utag;
    };
```

Special Topics 7

This chapter covers the following special topics:

- Open arrays
- Data type conversion
- Automatic binding
- Servers that export multiple interface versions
- Servers that contain multiple managers

The examples in this chapter omit most error-handling code and use ellipses (. . .) to indicate substantial omissions.

7.1 Open Arrays

DECrpc supports fixed arrays, which have an explicitly declared length, and open arrays, which have no explicitly declared length. Because the length of an open array is not known until run time, special treatment is required to dynamically inform stubs about the array length.

This section describes the NIDL constructs associated with open arrays and discusses the interface definition, client module, and manager module for a simple `primes` application that generates prime numbers and passes an open array as input and output.

7.1.1 NIDL Attributes for Arrays

NIDL provides two field attributes that apply only to arrays: `last_is` and `max_is`. These attributes identify *last* and *max* fields that, at run time, contain information about the length of an array. The client stub and server stub use the *last* and *max* information to marshall, unmarshall, and store the array.

An array with `last_is` or `max_is` must be either a member of a structure or a parameter of an operation. The attributes precede the array name in a *field_attribute_list*:

type_specifier [*field_attribute_list*] *array_declarator* [*array_length*]

The *array_length* is optional. To specify an open array, supply an asterisk (`*`) as the *array_length* or omit the *array_length* altogether. The *field_attribute_list* comprises one or both of the following elements, separated by commas:

```
last_is ( last )
max_is ( max )
```

7.1.1.1 The last_is Attribute – The `last_is` attribute enables client and server programs to indicate dynamically the size of an array. This attribute informs the NIDL Compiler that, at run time, *last* will be the index of the last array element to be passed. When an array passes from client to server, the client program assigns a value for *last*, and the client stub uses this value to marshall the array. Likewise, when an array passes from server to client, the server manager code assigns a value for *last*, and the server stub uses this value to marshall the array.

Note that *last* is an index, not a count.

The `last_is` attribute is required for open arrays. For a fixed array, `last_is` is not required, but you can use it to increase efficiency when you intend to pass only part of the array; the stubs will not marshall any element with an index greater than *last*. Examples 7-4 and 7-6 apply `last_is` to fixed arrays.

An array with `last_is` can appear either in the parameter list of an operation declaration or in the declaration of a structure. In an operation declaration, the array and its *last* are parameters of the operation; in a structure declaration, the array and its *last* are members of the structure, and the array must be the last member.

The following declaration specifies that, at run time, `nlast` will be the index of the last element to be passed in the array `narray`:

```
typedef struct {
    int nlast;
    char [last_is (nlast)] narray[];
    } name;
```

If an array has a *last*, the stub that sends the array uses the *last* to determine how many elements to marshall, and it embeds the element count in the transmitted representation of the array. The stub that receives the array uses this embedded count to determine how many elements it should unmarshall. Therefore, the *last*, whether a structure member or a parameter, must be available to the sending stub but need not be available to the receiving stub.

If the array and its *last* are members of a structure, this condition is automatically met because the array and the *last* are always sent together. However, if the array and its *last* are parameters of an operation, you must ensure that the *last* parameter travels with or before the array parameter: an in array requires an in *last*, but an out array can have either an in or an out *last*.

It is possible for a *last* to serve as both *last* and *max* for an array, as described in the next section.

7.1.1.2 The max_is Attribute – The max_is attribute enables a client program to indicate dynamically the maximum possible size of an array. This attribute informs the NIDL Compiler that, at run time, *max* will be the maximum possible index of the array. The client program assigns the value of *max*; the server stub uses this value when it allocates storage for the "surrogate" copy of the array on the server side.

Like *last*, *max* is an index, not a count.

You typically apply max_is to open arrays that are returned by the server, but you can omit it. If you omit max_is for an open array, the NIDL Compiler uses the *last* of the array as its *max*, as though you had declared max_is(*last*).

Like last_is, max_is can appear in an operation declaration or in a structure declaration. In an operation declaration, the array and its *max* are parameters of the operation; in a structure declaration, the array and its *max* are members of the structure, and the array must be the last member.

The following declaration specifies both max_is and last_is attributes for the array parray:

```
typedef struct {
    int pmax;
    int plast;
    int [max_is(pmax), last_is(plast)] parray[];
    } pixels;
```

Because the client program supplies *max* for use by the server stub, *max* must pass from client to server and therefore must have at least the in attribute. If you omit the max_is attribute and allow a *last* to serve as a *max*, this directional requirement applies to the *last*.

One implication of this is that a structure containing an open array can never be simply an out. If you intend the array to pass in the out direction only, the interface definition must declare the structure as both in and out, and the client program must set the input value of *last* to prevent the client stub from marshalling data; in the C syntax of NIDL, arrays are zero-based, so the input value of *last* should be –1.

7.1.2 The primes Interface Definition

Example 7-1 shows the NIDL definition for the `primes` interface. This definition contains only one declaration, that of the `primes$gen` operation. The operation passes input and output in the array `values`.

Example 7-1: The primes.idl Interface Definition

```
%c
[uuid(443d5a1a4000.0d.00.00.fe.da.00.00.00), version(1)]
interface primes
{
[idempotent]
void primes$gen(
    handle_t     [in] h,
    int          [in, out] *last,
    int          [in] max,
    status_$t    [comm_status, out] *st,
    int          [in, out, last_is(last), max_is(max)] values[]
    );
    /* the first element of values[] will be used
       to hold an input parameter */
}
```

The empty brackets indicate that `values` is an open array. The array, its *last*, and its *max* are all parameters of the `primes$gen` operation.

This interface definition also illustrates use of the `comm_status` parameter attribute. If a communications error occurs during a `primes$gen` call, a cleanup handler inserted by the NIDL Compiler in the client stub handles the error and passes the error code to the client in the `st` status parameter. Chapter 5 discusses status parameters.

7.1.3 The primes Client Module

Example 7-2 shows excerpts from the client module, `client.c`.

The client initializes `values` to a length of 1000 elements. It asks the user to specify the integer up to which prime numbers will be generated, and it assigns this integer to the first element of `values`.

The client sets `last` to 0, so that only one element will pass as input to the server. When it calls `primes$gen`, the client supplies 999 as the `max` parameter, to ensure that, on return, the array will not exceed the space allocated for it.

When `primes$gen` returns, the client prints the array elements, whose indexes range from 0 to `last`.

Example 7-2: Excerpts from the client.c Module for primes

```
...
#define MAXVALS 1000
...
main()
{
    handle_t h;
    status_$t st;
...
    ndr_$long_int values[MAXVALS], last;
    char buf[100];
    int i;
...
    printf("Generate primes up to what integer: ");
    gets(buf);
    values[0] = (ndr_$long_int)atoi(buf);
    last = 0;    /* marshall only the first element of the array */
    primes$gen(h, &last, MAXVALS-1, &st, values);
...
    printf("Primes are:\n");
    for (i = 0; i <= last; i++) printf("%d ", values[i]);
    printf("\n");
}
```

7.1.4 The primes Manager Module

Example 7-3 shows the manager module, `manager.c`.

The manager routine `primes$gen` checks integers for primeness and assigns prime numbers to elements of `values`. It quits when it reaches the limit specified on input by the client or when it reaches the array element with index `max`. Before it returns, `primes$gen` sets `last` to the index of the last element in `value`.

Example 7-3: The manager.c Module for primes

```
#include "primes.h"

globaldef primes_v1$epv_t primes_v1$manager_epv  {primes$gen};

void primes$gen(h, last, max, status, values)
handle_t h;
status_$t *status;
ndr_$long_int *last, max, values[];
{
    ndr_$long_int n, highest = values[0], index = 0;

    for (n = 2; n <= highest; n++)
        if (is_prime(n)) {
            values[index++] = n;
            if (index > max) break;
        }
    *last = index-1;
```

Example 7-3: (continued)

```
    status->all = status_$ok;
    return;
}

static int is_prime(n)
ndr_$long_int n;
{
    int i;

    for (i = n/2; i > 1; i--)
        if (i*(n/i) == n) return 0;
    return 1;
}
```

The xmitas and sparse applications, described in Section 7.2, apply last_is to fixed arrays and also show how to pass an array as a member of a structure.

7.2 Data Type Conversion

The NIDL transmit_as attribute lets you associate a **transmitted type** that stubs pass over the network with a **presented type** that clients and servers manipulate. You write routines to convert between the presented and transmitted types, and you link those routines with the stubs. This section lists the requirements for the conversion routines and presents two examples: one that uses type conversion to pass a complex data type and one that uses type conversion for efficiency. Chapter 4 describes the use of transmit_as in NIDL definitions.

7.2.1 Type Conversion Routines

When you associate a transmitted type with a presented type, you must write four routines to perform conversion and to manage storage for the types. This section specifies C prototypes for these routines; in the prototypes, PRES is the name of the presented type and TRANS is the name of the transmitted type. The PRES_to_xmit_rep routine allocates storage for the transmitted type and converts from the presented type to the transmitted type:

```
void PRES_to_xmit_rep (presented,transmitted)
    PRES presented;
    TRANS **transmitted;
```

The PRES_from_xmit_rep routine allocates storage for the presented type and converts from the transmitted type to the presented type:

```
void PRES_from_xmit_rep (transmitted, presented)
```

```
    TRANS *transmitted;
    PRES *presented;
```

The `PRES_free` routine frees any storage that has been allocated for the presented type by `PRES_from_xmit_rep`:

```
void PRES_free (presented)
    PRES presented;
```

The `PRES_free_xmit_rep` routine frees any storage that has been allocated for the transmitted type by `PRES_to_xmit_rep`:

```
void PRES_free_xmit_rep (transmitted)
    TRANS *transmitted;
```

7.2.2 Using Type Conversion to Pass Complex Types

The NIDL Compiler cannot generate stub code to marshall and unmarshall complex types such as trees, linked lists, and structures that contain pointers. Any data type containing a pointer not ''at top level'' is complex.

The `xmitas` application uses type conversion to pass a linked list as an open array. The client and server manipulate the linked list type. The client and server stubs transmit arrays over the network.

7.2.3 The xmitas Interface Definition

Example 7-4 shows the NIDL definition for the `xmitas` interface.

Example 7-4: The xmitas.idl Interface Definition

```
%c
[uuid(441f8a28a000.0d.00.00.fe.da.00.00.00), version(1)]
interface xmitas
{
    const int MAXELEMENTS = 100;    /* maximum size of list */

    typedef struct {
        int last;
        int [last_is(last)] values[MAXELEMENTS];
    } trans_t;

    typedef [transmit_as(trans_t)] struct {
        int value;
        list_t *next;
    } list_t;

    [idempotent]
        int xmitas$sum(handle_t [in] h, list_t [in] list);
}
```

The transmitted type, `trans_t`, is a structure whose members are the integer `last` and the integer array `values`. Though `values` has a declared length, the `last_is` attribute is supplied so that no more elements than necessary are passed.

The presented type, `list_t`, is a linked list structure whose members are the integer `value` and the pointer `next`, which points to the next `list_t`.

There is one operation in the `xmitas` interface, `xmitas$sum`. Its inputs are `h` (a handle) and `list` (a linked list). The operation returns an integer that is the sum of the values in `list`.

7.2.4 The xmitas util.c Module

Example 7-5 shows the `util.c` module, which contains routines to convert between the `list_t` and `trans_t` types and to allocate and free storage for those types.

Example 7-5: The util.c Module for xmitas

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

static void free_list_recursively();     /* auxiliary function */

void list_t_to_xmit_rep(list, xmit_struct) 1
list_t list;
trans_t **xmit_struct;
{
    int count = 0;
    list_t *lp = &list;

    /* allocate the structure */
    *xmit_struct = (trans_t *)malloc(sizeof(trans_t));

    /* copy the values from the list to the array */
    while (lp) {
        (*xmit_struct)->values[count++] = lp->value;
        lp = lp->next;
    }
    (*xmit_struct)->last = (ndr_$long_int)(count-1);
}

void list_t_from_xmit_rep(xmit_struct, list) 2
trans_t *xmit_struct;
list_t *list;
{

    int index = 0;

    /* reconstruct the linked list from the array */
    do {
        list->value = xmit_struct->values[index++];
        if (index <= xmit_struct->last)
            list->next = (list_t *)malloc(sizeof(list_t));
        else list->next = NULL;
```

Example 7-5: (continued)

```
        list = list->next;
    } while (index <= xmit_struct->last);
}

void list_t_free(list) [3]
list_t list;
{
    free_list_recursively(list.next);
}

void list_t_free_xmit_rep(xmit_struct) [4]
trans_t *xmit_struct;
{
    free(xmit_struct);
}

static void free_list_recursively(l)
list_t *lp;
{
    if (lp->next) free_list_recursively(lp->next);
    free(lp);
}

char *error_text(st)
status_$t st;
{
    static char buff[200];
    extern char *error_$c_text();
    return (error_$c_text(st, buff, sizeof buff));
}
```

[1] The first routine, `list_t_to_xmit_rep`, allocates storage for the structure to be transmitted and then copies values from the linked list into the array. It sets `(*xmit_struct)->last` to the index of the last element that it copied to `(*xmit_struct)->values`.

[2] The second routine, `list_t_from_xmit_rep`, copies values from the transmitted array into the linked list, allocating additional storage as it builds the list, until it reaches the array element with index `last`.

[3] Any storage allocated by `list_t_from_xmit_rep` for the linked list is freed by `list_t_free`.

[4] Any storage allocated by `list_t_to_xmit_rep` is freed by `list_t_free_xmit_rep`.

7.2.5 Using Type Conversion for Efficiency

The `sparse` application uses type conversion to transmit arrays in a run-length-encoded format. The code supplies routines to encode and decode the arrays. The stubs present sparse arrays to the client and server but pass compact arrays over the network.

This section discusses the interface definition and `util.c` module for `sparse`.

7.2.5.1 The sparse Interface Definition – Figure 7-6 shows the NIDL definition for the `sparse` interface.

Example 7-6: The sparse.idl Interface Definition

```
%c
[uuid(442548088000.0d.00.00.fe.da.00.00.00), version(1)]
interface sparse
{
    const int ARRAY_SIZE = 1000;
    const int CARRAY_SIZE = 2000;[1]
    /* worst case: twice the original size */

    /* a run-length-encoded representation of an array */

    typedef struct {
        int last;
        int [last_is(last)] data[CARRAY_SIZE];[2]
    } compress_t;

    /* this type will be transmitted as a more compact type */
    typedef [transmit_as(compress_t)] int compress_array[ARRAY_SIZE];[3]

    /* this type will be transmitted as is */
    typedef int nocompress_array[ARRAY_SIZE];[4]

    [idempotent]
        int sparse$compress_sum([5]
            handle_t [in] h,
            compress_array [in] array
            );

    [idempotent]
        int sparse$nocompress_sum([6]
            handle_t [in] h,
            nocompress_array [in] array
            );
}
```

[1] In the worst case, encoding doubles the length of an array, so the declared length of the compact array is twice that of the sparse array.

[2] Because we expect the compact array to be shorter we give it the `last_is` attribute and embed it in the `compress_t` structure with a `last`.

3 The example declares two sparse array types: `compress_array` has `compress_t` as its transmitted form.

4 The array `nocompress_array` is transmitted unchanged.

5 Both of the operations in the `sparse` interface take a sparse array as input and return the sum of its elements. The operation `sparse$compress_sum` passes its inputs in a compact array.

6 The operation `sparse$nocompress_sum` passes a sparse array.

7.2.5.2 The sparse util.c Module – Example 7-7 shows the `util.c` module, which contains the conversion routines for the `sparse` application. These routines are similar to those for the `xmitas` application.

Example 7-7: The util.c Module for sparse

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

void compress_array_to_xmit_rep(array, xmit_struct) 1
compress_array array;
compress_t **xmit_struct;
{
    int rep, val, index = 0, pos = 0;

    /* allocate the structure */
    *xmit_struct  (compress_t *)malloc(sizeof(struct compress_t));

    /* run-length encode the array */
    do {
        rep = 0;
        val = array[pos];
        while (pos < ARRAY_SIZE && array[pos] == val) {
            pos++;
            rep++;
        }
        (*xmit_struct)->data[index] = rep;
        (*xmit_struct)->data[index+1] = val;
        index += 2;
    } while (pos < ARRAY_SIZE);

    (*xmit_struct)->last = index-1;2
}

void compress_array_from_xmit_rep(xmit_struct, array) 3
compress_t *xmit_struct;
compress_array *array;
{
    int index, rep, count = 0;

    for (index = 0; index < xmit_struct->last; index+=2)
        for (rep = 0; rep < xmit_struct->data[index]; rep++)
            (*array)[count++] = xmit_struct->data[index+1];
}
```

Example 7-7: (continued)

```
void compress_array_free(object) [4]
compress_array object;
{
    /* no freeing is appropriate here */
}

void compress_array_free_xmit_rep(xmit_struct) [5]
compress_t *xmit_struct;
{
    free(xmit_struct);
}

char *error_text(st)
status_$t st;
{
    static char buff[200];
    extern char *error_$c_text();

    return (error_$c_text(st, buff, sizeof buff));
}
```

[1] The `compress_array_to_xmit_rep` routine allocates storage for the compact array and then encodes the sparse array.

[2] The routine sets `(*xmit_struct)->last` to the index of the last element that it copied to `(*xmit_struct)->data`, so that no more elements are passed than necessary.

[3] The `compress_array_from_xmit_rep` routine decodes the compact array, reconstructing the sparse array. Storage for the sparse array has already been allocated, so this routine does not perform any allocation.

[4] Because `compress_array_from_xmit_rep` did not allocate any storage, `compress_array_free` does not need to free any and thus is defined as a null operation.

[5] Storage allocated by `compress_array_to_xmit_rep` is freed by `compress_array_free_xmit_rep`.

7.2.5.3 Restrictions – You cannot use a data type with the `transmit_as` attribute as an element of an array or as a member of a structure or union. In effect, you can use a type with `transmit_as` only as an operation parameter.

A data type with the `transmit_as` attribute cannot serve as the transmitted type for another type.

7.3 Automatic Binding

Automatic binding allows a client to represent objects with generic handles rather than RPC handles. The data type of a generic handle must have the `handle` type attribute. The generic handle can be either a first parameter in each operation (an explicit handle) or a global variable in the client (an implicit handle).

Because the RPC run-time library uses only RPC handles, you must supply an autobinding routine that generates RPC handles from generic handles. The client stub invokes the autobinding routine each time the client makes a remote procedure call. In addition, you supply an autounbinding routine that performs any necessary cleanup (for instance, freeing the RPC handle) after the remote call returns.

7.3.1 Automatic Binding Activity

If an application uses automatic binding, the following occurs when the client makes a remote procedure call:

1. The client makes a remote procedure call, through the client switch, to the stub. The client provides a generic handle either as the first parameter of the call (an explicit handle) or through a global variable (an implicit handle).
2. The stub calls the autobinding procedure, passing to it the generic handle.
3. The autobinding procedure returns an RPC handle to the stub.
4. The stub uses the RPC handle as a parameter to the `rpc_$sar` library routine.
5. The `rpc_$sar` routine returns the server response to the stub.
6. The stub calls the autounbinding procedure, passing to it the RPC handle.
7. The autounbinding procedure frees the RPC handle and any unneeded resources associated with the generic handle.
8. The stub returns to the client.

7.3.2 Autobinding and Autounbinding Routines

When you use a generic handle type, you must write autobinding and autounbinding routines. This example shows the autobinding routine for UUIDs from the bank example. (Example 7-8 shows the entire routine.) The routine generates an RPC handle from an object UUID and returns the RPC handle:

```
handle_t uuid_$t_bind(object)
uuid_$t object;
```

The next examples show C prototypes for these routines; in the prototypes, *GENERIC* is the name of the generic handle type (replacing `uuid_$t` in the previous example). The autobinding routine *GENERIC*`_bind` generates an RPC handle from a generic handle and returns the RPC handle:

```
handle_t GENERIC_bind (g-handle)
GENERIC g-handle ;
```

The autounbinding routine *GENERIC*`_unbind` takes two inputs, a generic handle and the RPC handle that was generated from it, and has no outputs:

```
void GENERIC_unbind (g-handle, rpc-handle)
GENERIC g-handle;
handle_t rpc-handle );
```

An autounbinding routine typically frees the RPC handle and any unneeded resources associated with the generic handle, but it is not required to do anything.

7.3.3 Automatic Binding in the bank Example

Examples 7-8 and 7-9 show the autobinding and autounbinding routines from the `bank` example.

These routines, defined in the `uuidbind.c` module, enable the `bank` example to use UUIDs as generic handles. They maintain a cache of handles to save the expense of invoking `lb_$lookup_object` and `rpc_$bind` every time the client makes a remote procedure call; this approach is particularly useful in applications where the client tends to make several calls to access the same object. The file `RPC$IDL:NBASE.IDL` defines the UUID data type, `uuid_$t`, and assigns to this type the `handle` type attribute.

7.3.3.1 The bank Autobinding Routine – The autobinding routine, `uuid_$t_bind`, searches the cache for an RPC handle that matches the generic handle (the object UUID). If there is no matching handle in the cache, it calls `lb_$lookup_object` to get the location of the object and calls `rpc_$bind` to create a new handle. It uses `rpc_$dup_handle` to return a copy of the handle.

Each handle in the cache has an associated reference count. When all copies of a handle have been freed, meaning that its binding is not in use, the original handle is kept available but is considered ''collectible.'' If its entry in the cache is needed for a new handle, it can be freed.

Example 7-8: An Autobinding Routine for UUIDs

```
/*
 * Table mapping UUIDs into RPC handles.
 */

static struct db_entry {
    boolean  valid;           /* Is this entry valid? */
    uuid_$t  obj;             /* Object UUID */
    handle_t handle;          /* RPC handle for the object */
    unsigned short refcnt;    /* # of references on this entry */
} uuid_db[MAX_ENTRIES];

/*
 * Autobinding procedure for type "uuid_$t".
 */

handle_t uuid_$t_bind(object)
uuid_$t object;
{
    short i, invalid_i = -1, collectible_i = -1;
    lb_$entry_t lb_entry;
    unsigned long n_results;
    status_$t st;
    lb_$lookup_handle_t lookup_handle = lb_$default_lookup_handle;

    /*
     * Scan the table for an entry that has a matching UUID.  If
     * we find one, return the handle that's stored there.  While
     * scanning, keep note of the last invalid entry (i.e. one that
     * is unused) and the last collectible entry (i.e. one that has
     * an object and handle but isn't being referenced by anyone).
     */
    for (i = 0; i < MAX_ENTRIES; i++) {
        struct db_entry *db = &uuid_db[i];
        if (! db->valid)
            invalid_i = i;
        else {
            if (bcmp(&db->obj, &object, sizeof object) == 0) {
                db->refcnt++;
                return (rpc_$dup_handle(db->handle, &st));
            }
            if (db->refcnt == 0)
                collectible_i = i;
        }
    }
    /*
     * Didn't find a match in the table.
     * Ask the LB for the location.
     */
    lb_$lookup_object(&object, &lookup_handle, 1L, &n_results,
            &lb_entry, &st);
    if (st.all != status_$ok || n_results <= 0) {
        fprintf(stderr,
                "(uuid_$t_bind) Lookup failed, n_results%ld\n",
                n_results);
```

Example 7-8: (continued)

```
        pfm_$signal(st);
    }

    /*
     * Decide whether we have an entry to use.
     * Free the current handle if we're collecting the entry.
     */
    if (invalid_i != -1)
        i = invalid_i;
    else if (collectible_i != -1) {
        i=  collectible_i;
        rpc_$free_handle(uuid_db[i].handle, &st);
    }
    else {
        fprintf(stderr, "(uuid_$t_bind) No space in cache\n");
        abort();
    }

    /*
     * Fill in the entry with our values.
     */
    uuid_db[i].obj    = object;
    uuid_db[i].valid  = true;
    uuid_db[i].refcnt = 1;

    /*
     * Make an RPC handle for the object and location and return it.
     */
    uuid_db[i].handle  rpc_$bind(&object, &lb_entry.saddr,
            lb_entry.saddr_len, &st);
    if (st.all != status_$ok)
        pfm_$signal(st);
    return (rpc_$dup_handle(uuid_db[i].handle, &st));
}
```

7.3.3.2 The bank Autounbinding Routine – The autounbinding routine, `uuid_$t_unbind`, uses `rpc_$free_handle` to free a copy of the RPC handle that matches the generic handle and decrements the reference count of the generic handle.

Example 7-9: An Autounbinding Routine for UUIDs

```
/*
 * Autounbinding procedure for type "uuid_$t".
 */
void uuid_$t_unbind(object, handle) [1]
uuid_$t object;
handle_t handle;
{
    unsigned short i;
    status_$t st;

    /*
     * Scan the table looking for the handle.
     */
    for (i = 0; i < MAX_ENTRIES; i++) {
        struct db_entry *db = &uuid_db[i];

        if (db->valid && db->handle == handle) {
            rpc_$free_handle(handle, &st); [2]
            db->refcnt--; [3]
            return;
        }
    }

    fprintf(stderr,
        "(uuid_$t_bind) tried to free a handle we didn't return\n");
    abort();
}
```

[1] The autounbinding routine `uuid_$t_unbind` takes two arguments–an object (of type `uuid_t$` and a handle of type `handle_t`.

[2] The routine uses `rpc_$free_handle` to free a copy of the RPC handle that matches the generic handle.

[3] The routine then decrements the reference count of the handle.

7.4 Multiple Interface Versions

DECrpc allows a single server to simultaneously export several versions of an interface. The `binopmv` example, an extension of the `binop_lu` example described in Chapter 3, illustrates this feature.

There are two versions of the `binopmv` interface. The first version is essentially identical to the `binop_lu` interface; the second version has one additional operation.

The `binopmv` example actually does not require a server that exports both versions of the interface. Chapter 5 describes a way to add operations to interfaces while maintaining backward compatibility. However, `binopmv` illustrates the most general way to compatibly modify an interface.

This section describes the interface definitions, the client modules, the server module, and the manager module for binopmv.

7.4.1 The binopmv Interface Definitions

The binopmv example has two interface definition files, named vers1.idl and vers2.idl.

7.4.1.1 The vers1.idl Interface Definition – Example 7-10 shows vers1.idl, the NIDL definition for version 1 of the binopmv interface. This interface definition declares one operation, binopmv$add.

Example 7-10: The vers1.idl Interface Definition for binopmv

```
%c
[uuid(4433af7ed000.0d.00.00.fe.da.00.00.00), version(1)]
interface binopmv
{
[idempotent]
    void binopmv$add(
        handle_t [in] h,
        long [in] a,
        long [in] b,
        long [out] *c
        );
}
```

7.4.1.2 The vers2.idl Interface Definition – Example 7-11 shows vers2.idl, the NIDL definition for version 2 of the binopmv interface. The definitions for the two versions of binopmv specify the same interface UUID and the same interface name, but different version numbers.

The definition for version 2 declares two operations, binopmv$add and binopmv$sub.

Example 7-11: The vers2.idl Interface Definition for binopmv

```
%c
[uuid(4433af7ed000.0d.00.00.fe.da.00.00.00), version(2)]
interface binopmv
{
[idempotent]
    void binopmv$add(
        handle_t [in] h,
        long [in] a,
        long [in] b,
        long [out] *c
        );
[idempotent]
```

Example 7-11: (continued)

```
    void binopmv$sub(
        handle_t [in] h,
        long [in] a,
        long [in] b,
        long [out] *c
        );
}
```

7.4.2 Compiling the Interface Definitions

When you compile interface definitions for an application whose server will export multiple interface versions, you must specify the NIDL Compiler's `-m` qualifier.

If invoked with `-m`, the NIDL Compiler appends the version number to the interface name when it generates identifiers in the stub and header files. In effect, different versions of an interface have different names.

The `nidl` reference page describes all of the NIDL Compiler qualifiers.

Table 7-1 lists the identifiers that the NIDL Compiler generates for the `binopmv` application. These identifiers are generated from the interface name and the version number.

Table 7-1: Identifiers in the binopmv Example

Component	Identifier for Version 1	Identifier for Version 2
EPV type	`binopmv_v1$epv_t`	`binopmv_v2$epv_t`
Client EPV	`binopmv_v1$client_epv`	`binopmv_v2$client_epv`
Server EPV	`binopmv_v1$server_epv`	`binopmv_v2$server_epv`
Interface specifier	`binopmv_v1$if_spec`	`binopmv_v2$if_spec`

7.4.3 The binopmv Client Modules

There are two client programs. The first, `client1.c`, uses version 1 of the interface and calls `binopmv$add`. The second, `client2.c`, uses version 2 of the interface and calls both `binopmv$add` and `binopmv$sub`.

In most respects, the `client1.c` and `client2.c` programs are similar to the `binop_lu` client described in Chapter 3, so the following discussions concentrate on the client program's use of multiple interface versions.

7.4.3.1 Header Files – Each client includes the header file for its version of the interface as shown in the following examples.

This example shows the include file for `client1.c`:

```
#include "vers1.h"
```

This example shows the include file for `client2.c`:

```
#include "vers2.h"
```

7.4.3.2 Location Broker Lookup Criteria – The clients perform Location Broker lookups by interface. Each client supplies to `lb_$lookup_interface` the `id` member of the `if_spec` for its version of the interface.

This example shows the `lb_$lookup_interface` call for `client1.c`:

```
lb_$lookup_interface(&binopmv_v1$if_spec.id, &lookup_handle, 1L,
                     &nresults, &entry, &st);
```

This example shows the `lb_$lookup_interface` call for `client2.c`:

```
lb_$lookup_interface(&binopmv_v2$if_spec.id, &lookup_handle, 1L,
                     &nresults, &entry, &st);
```

Although these lookup calls appear to be different, they are in effect identical because versions 1 and 2 of the interface have the same UUID. Hence, the lookup calls will return information about all servers for `binopmv`, regardless of version. Each client must either check that a server exports the correct version or deal with possible version mismatches.

7.4.3.3 Checking Interface Versions – After a `binopmv` client has obtained the Location Broker entry for a `binopmv` server, the client binds its handle to the location of the server and then checks that the server exports a matching version of the interface. Example 7-12 shows the version-checking code in `client1.c`; `client2.c` contains essentially the same code.

Example 7-12: Version-Checking Code in the client1.c Module for binopmv

```
...
#include "vers1.h"
...
#define VERSION 1    /* version of interface requested */
...
    handle_t h;
    status_$t st;
    rrpc_$interface_vec_t ifs;
```

Example 7-12: (continued)

```
    unsigned long lastif;
    int k, passes, found_version;
...
    /* check for appropriate version */

    rrpc_$inq_interfaces(h, 2L, ifs, (ndr_$long_int *)&lastif, &st); [1]

    for (k = 0, found_version = 0; k <= lastif; k++) [2]
        if (ifs[k].vers == VERSION) found_version = 1;
    if (!found_version) {
        fprintf(stderr, "Couldn't get version %d\n", VERSION);
        exit(1);
    }
    else printf("Found version %d\n", VERSION);
...
```

[1] The client calls `rrpc_$inq_interfaces` to obtain an `rrpc_$interface_vec_t` , an array of interface specifiers for the interfaces exported by the server.

[2] The client code checks the `vers` member of each interface specifier against its own version until it finds a match.

7.4.4 The binopmv Server Module

The server module, `server.c`, largely resembles the `binop_lu` server described in Chapter 3, but does all of its registrations and unregistrations twice, once for each interface version.

7.4.4.1 Registrations and Unregistrations – Example 7-13 shows the registration and unregistration code in `server.c`.

Example 7-13: Registrations and Unregistrations in the server.c Module for binopmv

```
...
#include "vers1.h" [1]
#include "vers2.h"

globalref uuid_$t uuid_$nil;
globalref binopmv_v1$epv_t binopmv_v1$manager_epv; [2]
globalref binopmv_v2$epv_t binopmv_v2$manager_epv;
...
    status_$t st;
    socket_$addr_t loc;
    unsigned long llen;
    lb_$entry_t lb_entry[2];
    pfm_$cleanup_rec crec;
```

Example 7-13: (continued)

```
...
    /* register version 1... */

    rpc_$register_mgr(&uuid_$nil, &binopmv_v1$if_spec, [3]
         binopmv_v1$server_epv,
         (rpc_$mgr_epv_t)&binopmv_v1$manager_epv, &st);

    /* ...and version 2 with the run-time library */

    rpc_$register_mgr(&uuid_$nil, &binopmv_v2$if_spec, [4]
         binopmv_v2$server_epv,
         (rpc_$mgr_epv_t)&binopmv_v2$manager_epv, &st);

    /* register version 1 with the lb */

    lb_$register(&uuid_$nil, &uuid_$nil, &binopmv_v1$if_spec.id, 0L, [5]
         (ndr_$char *)"binopmv example (v1)", &loc, llen,
         &lb_entry[0], &st);

    /* ...and version 2 with the lb */

    lb_$register(&uuid_$nil, &uuid_$nil, &binopmv_v2$if_spec.id, 0L, [6]
         (ndr_$char *)"binopmv example (v2)", &loc, llen,
         &lb_entry[1], &st);

    st = pfm_$cleanup(&crec); [7]
    if (st.all ! pfm_$cleanup_set) {
        status_$t stat;
        fprintf(stderr, "Server received signal - %s\n",
            error_text(st));
        lb_$unregister(&lb_entry[0], &stat);
        lb_$unregister(&lb_entry[1], &stat);
        rpc_$unregister(&binopmv_v1$if_spec, &stat);
        rpc_$unregister(&binopmv_v2$if_spec, &stat);
        pfm_$signal(st);
    }
...
```

[1] The server includes the header files for both versions of the interface.

[2] The server declares two manager DPVs as external variables.

These EPVs are defined in the manager module. Their names resemble those of the client and server EPVs, but this is merely by convention. Manager EPV names are arbitrary, since they appear only in server and manager code that you write, not in code that the NIDL Compiler generates.

[3] Because it exports several interface versions, the `binopmv` server must register each of its manager versions with the RPC run-time library at its (the server's) host. These registrations enable the run-time library to dispatch incoming requests to the correct version of the manager.

[4] This call registers the second version with the run-time library.

[5] The server also registers twice with the Location Broker. These registrations supply the same UUID to the Location Broker, and hence are indistinguishable to a client performing lookups. Each entry has a different annotation.

[6] This call registers the second version with the Location Broker.

[7] Before it calls `rpc_$listen` to begin accepting requests, the server sets a cleanup handler. If it is signaled, the server removes all of its registrations before it exits.

7.4.5 The binopmv Manager Module

Figure 7-14 shows `manager.c`, the manager module for `binopmv`. This module contains all the code to implement both versions of `binopmv`.

Example 7-14: The manager.c Module for binopmv

```
#include "vers1.h" [1]
#include "vers2.h"

globaldef binopmv_v1$epv_t binopmv_v1$manager_epv =  [2]
    {binopmv$add}; [3]
globaldef binopmv_v2$epv_t binopmv_v2$manager_epv  = [4]
    {binopmv$add, binopmv$sub}; [5]

void binopmv$add(h, a, b, c)
handle_t h;
ndr_$long_int a, b, *c;
{
    *c  a + b;
}

void binopmv$sub(h, a, b, c)
handle_t h;
ndr_$long_int a, b, *c;
{
    *c  a - b;
}
```

[1] The manager includes both versions of the header file.

[2] This global definition defines the manager EPV for version 1.

[3] The EPV for version 1 lists only one operation.

4 This global definition defines the manager EPV for version 2.

5 The EPV for version 2 lists two operations.

7.4.6 Changing Operations in Interfaces with Multiple Versions

In the `binopmv` application, version 1 and version 2 can share the manager routine for `binopmv$add` because the operation is identical in the two versions. If an operation has different signatures or implementations in two versions of the interface, you must write two manager routines for the operation.

Suppose you are changing the implementation of `binopmv$add` between versions 1 and 2, and you are building a server that exports both versions. You must give distinct names such as `binopmv_v1$add` and `binopmv_v2$add` to the two versions of the manager routine. Because these names are not declared in the `vers1.h` and `vers2.h` header files that the NIDL Compiler generates, you must declare them in the manager module.

Example 7-15 shows what a `binopmv` manager with two versions of `binopmv$add` might look like.

Example 7-15: A Manager Module with Two Versions of an Operation

```
#include "vers1.h"
#include "vers2.h"

void binopmv_v1$add();
void binopmv_v2$add();

globaldef binopmv_v1$epv_t binopmv_v1$manager_epv
    {binopmv_v1$add};
globaldef binopmv_v2$epv_t binopmv_v2$manager_epv
    {binopmv_v2$add, binopmv$sub};

void binopmv_v1$add(h, a, b, c)      /* "old implementation" */1
handle_t h;
ndr_$long_int a, b, *c;
{
    *c  a + b;
}

void binopmv_v2$add(h, a, b, c)      /* "new implementation" */1
handle_t h;
ndr_$long_int a, b, *c;
{
    *c = b + a;
}

void binopmv$sub(h, a, b, c)
handle_t h;
ndr_$long_int a, b, *c;
```

Example 7-15: (continued)

```
{
    *c = a - b;
}
```

[1] In this manager, the two versions of the add operation have different names and trivially different implementations. Clients of either interface version continue to invoke the operation by its name in the interface definition, `binopmv$add`.

Of course, if an operation has a different signature as well as a different implementation in two versions of an interface, the manager routines and the interface definitions must reflect this difference.

7.4.7 Constants and Types in Interfaces with Multiple Versions

When you define a manager EPV, you can declare either that two versions of an interface will share a manager routine (as in Example 7-14) or that they will use different manager routines (as in Example 7-15). Thus, the names of the manager routines in a server will not conflict. The names of constants and types, however, can conflict.

If you declare the same type in two versions of an interface definition, the NIDL Compiler emits a C `typedef` declaration for the type in both of the C header files it generates. When you build a server program that exports both interface versions, the server includes both header files, and hence the type declarations are duplicated. Most C compilers reject such duplicate type declarations.

To avoid conflicts of type names, extract type declarations that are shared by the two versions of the interface and put these declarations in a version-independent interface definition that is imported by the two version-specific interface definitions. When you compile the definitions, the NIDL Compiler emits directives in the version-specific header files to include the version-independent header file.

In effect, a server that exports both versions of the interface includes this file twice, but every header file generated by the NIDL Compiler contains conditional statements to ensure that its contents are read only once, and therefore no declarations are duplicated.

If you declare a constant in two versions of an interface definition, the NIDL Compiler emits a C preprocessor `#define` directive for the constant in both of the C header files it generates. Although most C preprocessors accept the resulting duplication, it is better practice to define each constant only once, so we recommend that you keep shared constants together with shared types in a separate interface definition file. Example 7-16 shows what an interface definition file for shared types and constants might look like.

The interface requires a name but no attributes.

Example 7-16: An Interface Definition File for Shared Types and Constants

```
%c
interface sharedstuff
{
const VSIZE 1024;

typedef struct {
    int vlast;
    float [last_is(vlast)] varray [VSIZE];
    } values;
}
```

7.5 Multiple Managers

DECrpc allows one server to implement an interface for several object types. A separate manager implements each combination of interface and type. The server registers its objects and their types with the RPC run-time library and the Location Broker; it registers its managers with the RPC run-time library. This section describes the `stacks` application, in which a server manages two types of stacks, one based on lists and one based on arrays.

7.5.1 The stacks Interface Definition

Example 7-17 shows `stacks.idl`, the NIDL definition for the `stacks` interface. There are operations to initialize a stack, to push a value onto a stack, and to pop a value off a stack. Because the interface definition is purely syntactic, it does not indicate in any way the existence of two types of stacks. Different object types require different implementations of operations, but not different signatures.

When you compile `stacks.idl`, specify the NIDL Compiler's `-m` qualifier. The `nidl` reference description describes the NIDL Compiler qualifiers.

Example 7-17: The stacks.idl Interface Definition

```
%c
[uuid(4438675bf000.0d.00.00.fe.da.00.00.00), version(1)]
interface stacks
{
[idempotent]
    void stacks$init(
        handle_t [in] h
        );

/* stack functions return non-zero on error, zero otherwise */
```

Example 7-17: (continued)

```
int stacks$push(
    handle_t [in] h,
    int [in] value
    );

int stacks$pop(
    handle_t [in] h,
    int [out] *value
    );
}
```

7.5.2 The stacksdf.h Header File

Most of the examples in this book do not involve a particular object and hence specify `uuid_$nil` as the object identifier. The `bank` example, introduced to illustrate automatic binding, accesses two bank databases that are objects of the same type. The `stacks` example accesses two stacks that are objects of different types.

The `stacksdf.h` header file, shown in Example 7-18, defines symbolic constants to represent UUIDs for the two stacks (ASTACK and LSTACK) and their types (ASTACKT and LSTACKT). The replacement texts for these constants are C representations of UUIDs, which are generated by invoking `uuid_gen` with the `-C` qualifier.

Example 7-18: The stacksdf.h Header File

```
/* the two stack objects and their types */

/* the array-based object */
#define ASTACK {0x44349d2c, 0x2000, 0x0000, 0x0d, \
                     {0x00, 0x00, 0xfe, 0xda, 0x00, 0x00, 0x00}}

#define ASTACKT{0x44349e25, 0x0000, 0x0000, 0x0d, \
                     {0x00, 0x00, 0xfe, 0xda, 0x00, 0x00, 0x00}}

/* the list-based object */
#define LSTACK {0x44349e48, 0x2000, 0x0000, 0x0d, \
                     {0x00, 0x00, 0xfe, 0xda, 0x00, 0x00, 0x00}}

#define LSTACKT{0x44349eed, 0x6000, 0x0000, 0x0d, \
                      {0x00, 0x00, 0xfe, 0xda, 0x00, 0x00, 0x00}}
```

7.5.3 The stacks Client Module

Example 7-19 shows excerpts from the client module, `client.c`. The client program lets the user access both types of stacks within one session; it maintains a separate handle for each stack. (Other clients discussed maintain only one handle.) The handles are kept in an array, as are the UUIDs for the stack types. For each type, the client:

1. Performs a Location Broker lookup by type
2. Scans the entries returned for one with the desired interface and address family
3. Binds a handle to represent the object and the location registered in the entry

When the client program calls `stacks$push` or `stacks$pop`, the object UUID in the handle determines the stack to be accessed.

Example 7-19: Excerpts from the client.c Module for stacks

```
...
#include "stacks.h"
#include "stackdf.h"
...
#define MAXENTRIES 5    /* how many L.B. entries we can handle */
...
main()
{
    handle_t handle[2];
    status_$t st;
    lb_$entry_t entries[MAXENTRIES];
...
    static uuid_$t types[2] = {ASTACKT, LSTACKT};
    int s, t, k, found_if;
    ndr_$long_int val;
    char command[100], which[100], value[100];
...
    /* bind handles for each object type */
    for (t = 0; t < 2; t++) {
        /* find lb entries for the type */
        lb_$lookup_type(&types[t], &lookup_handle, MAXENTRIES, &nresults,
                entries, &st);
        if (nresults < 1) {
            fprintf(stderr,
                    "Couldn't find interfaces for type[%d]\n", t);
            exit(1);
        }

        /* check for appropriate interface for the type */
        for (k = 0, found_if = 0; k < nresults; k++)
            if (uuid_$equal(&entries[k].obj_interface,
                    &stacks_v1$if_spec.id) &&
                socket_$valid_family(entries[k].saddr.family,&st))
{
                found_if = 1; /* found appropriate interface */
                break;
            }
        if (!found_if) {
            fprintf(stderr, "Couldn't find appropriate interface\n");
```

Example 7-19: (continued)

```
            exit(1);
        }

        /* bind handle */
        handle[t] = rpc_$bind(&entries[k].object,
            &entries[k].saddr, entries[k].saddr_len, &st);
    }
    printf("Initialize stack objects (y/n)? ");
    gets(command);
    if (*command != ’n’ && *command != ’N’) {
        stacks$init(handle[0]);
        stacks$init(handle[1]);
    }

    do {
        printf("push, pop, or quit: ");
        gets(command);

        if (!strcmp(command, "quit")) break;

        printf("astack or lstack: ");
        gets(which);

        if (!strcmp(which, "astack")) s = 0;
        else s = 1;

        if (!strcmp(command, "push")) {
            printf("value: ");
            gets(value);
            val   (ndr_$long_int)atoi(value);
            printf("Pushing %d onto %s...",
                val, s?"lstack":"astack");
            if (stacks$push(handle[s], val)) printf("stack full!\n");
            else printf("successful\n");
        }
        else if (!strcmp(command, "pop")) {
            printf("Popping off of %s...", s?"lstack":"astack");
            if (stacks$pop(handle[s], &val))
                printf("nothing on stack!\n");
            else printf("value is %d\n", val);
        }
    } while (strcmp(command, "quit"));
}
```

7.5.4 The stacks Server Module

The `server.c` module is linked together with two manager modules to form the `stacks` server program, as shown in Example 7-20.

The `stacks` server offers access to both types of stacks. It registers the stack objects and types with the RPC run-time library and the Location Broker, and it registers its managers with the RPC run-time library.

The Location Broker registrations enable clients to look up the objects, types, and interfaces that the server supports, along with the location of the server.

Example 7-20: Registrations and Unregistrations in the server.c Module for stacks

```
...
#include "stackdf.h"
#include "stacks.h"
...
globalref stacks_v1$epv_t stacks_v1$amanager_epv; [1]
globalref stacks_v1$epv_t stacks_v1$lmanager_epv;
...
    status_$t st;
    lb_$entry_t lb_entry[2];
    pfm_$cleanup_rec crec;
    static uuid_$t astack = ASTACK, astackt = ASTACKT;
    static uuid_$t lstack = LSTACK, lstackt = LSTACKT;
...
    /* register manager and object for array-based stack object... */

    rpc_$register_mgr(&astackt, &stacks_v1$if_spec,  [2]
        stacks_v1$server_epv,
        (rpc_$mgr_epv_t)&stacks_v1$amanager_epv, &st);

    rpc_$register_object(&astack, &astackt, &st); [3]

    /* ...and list-based stack object with the run-time library */

    rpc_$register_mgr(&lstackt, &stacks_v1$if_spec, [4]
        stacks_v1$server_epv,
        (rpc_$mgr_epv_t)&stacks_v1$lmanager_epv, &st);
    rpc_$register_object(&lstack, &lstackt, &st);[5]

    /* register array-based stack object/interface... */

    lb_$register(&astack, &astackt, &stacks_v1$if_spec.id, 0L,
        (ndr_$char *)"astack example", &loc, llen, &lb_entry[0], &st); [6]
    /* ...and list-based stack object/interface with the lb*/

    lb_$register(&lstack, &lstackt, &stacks_v1$if_spec.id, 0L,
        (ndr_$char *)"lstack example", &loc, llen, &lb_entry[1], &st); [7]

    st = pfm_$cleanup(&crec); [8]
    if (st.all ! pfm_$cleanup_set) {
        status_$t stat;
        fprintf(stderr, "Server received signal - %s\n",
            error_text(st));
```

Example 7-20: (continued)

```
        lb_$unregister(&lb_entry[0], &stat); 9
        lb_$unregister(&lb_entry[1], &stat);
        rpc_$unregister(&stacks_v1$if_spec, &stat); /* once for each */
        rpc_$unregister(&stacks_v1$if_spec, &stat); /*    manager    */
        pfm_$signal(st);
    }
...
```

1 The server module declares two manager EPVs as external variables.

2 The manager registrations (`rpc_$register_mgr` calls) tell the RPC run-time library what combination of interface and type each manager implements. When the server receives a remote procedure call from a client, the run-time library dispatches the call to the correct manager. This first call registers the manager for the array-based stack object.

3 The object registrations (`rpc_$register_object` calls) tell the RPC run-time library what objects the server supports and what the type of each object is. This first call registers the array-based stack object.

4 The second manager registration registers the manager for the list-based stack object.

5 The second object registration registers the list-based stack object with the run-time library.

6 The Location Broker registrations enable clients to look up the objects, types, and interfaces that the server supports, along with the location of the server. This call registers the array-based stack object/interface with the Location Broker.

7 The second call to `lb_$register` registers the array-based stack object/interface.

8 Before it calls `rpc_$listen` to begin accepting requests, the server sets a cleanup handler.

9 If the cleanup handler is signaled, the server removes all of its registrations before it exits.

7.5.5 The stacks Manager Modules

A separate manager module implements the `stacks` interface for each type of stack: `lmanager.c` (Example 7-21) manages stacks based on linked lists, and `amanager.c` (Example 7-22) manages stacks based on arrays.

Each manager module defines a manager EPV. The EPV specifies the names under which the `stacks` operations are implemented. Because both managers are being linked in one server, the two implementations of each operation have different names.

Example 7-21: The lmanager.c Manager Module for stacks

```
#include "stacks.h"

void stacks$lstack_init();
ndr_$long_int stacks$lstack_push(), stacks$lstack_pop();

globaldef stacks_v1$epv_t stacks_v1$lmanager_epv =
    {stacks$lstack_init, stacks$lstack_push, stacks$lstack_pop};

#define NULL (struct node *)0
extern struct node *malloc();

static struct node {
    ndr_$long_int value;
    struct node *next;
} the_stack;

void stacks$lstack_init(h)
handle_t h;
{
    the_stack.next = NULL;
}

ndr_$long_int stacks$lstack_push(h, value)
handle_t h;
ndr_$long_int value;
{
    struct node *head = malloc(sizeof(struct node));
    if (head == NULL) return -1;            /* stack is full */

    head->value = value;
    head->next = the_stack.next;
    the_stack.next = head;
    return 0;
}

ndr_$long_int stacks$lstack_pop(h, value)
handle_t h;
ndr_$long_int *value;
{
    struct node *head = the_stack.next;
    if (head == NULL) return -1;           /* stack is empty */

    *value  head->value;
    the_stack.next = head->next;
    free(head);
    return 0;
}
```

Example 7-22: The amanager.c Manager Module for stacks/

```
#include "stacks.h"

void stacks$astack_init();
ndr_$long_int stacks$astack_push(), stacks$astack_pop();

globaldef stacks_v1$epv_t stacks_v1$amanager_epv
    {stacks$astack_init, stacks$astack_push, stacks$astack_pop};

#define STACKSIZE    1000

static struct {
    int head;
    ndr_$long_int values[STACKSIZE];
} the_stack;

void stacks$astack_init(h)
handle_t h;
{
    the_stack.head = STACKSIZE;
}

ndr_$long_int stacks$astack_push(h, value)
handle_t h;
ndr_$long_int value;
{
    if (the_stack.head == 0) return -1;         /* stack is full */

    the_stack.values[--the_stack.head]  value;

    return 0;
}

ndr_$long_int stacks$astack_pop(h, value)
handle_t h;
ndr_$long_int *value;
{

    if (the_stack.head == STACKSIZE) return -1; /* stack is empty */
    *value = the_stack.values[the_stack.head++];
    return 0;
}
```

error_$ Routine Reference Pages 8

This chapter contains reference pages for the `error_$` routines.

The error text database operations use the `error_$c_get_text` and `error_$c_text` library routines to convert status codes into textual error messages. The run-time library reports operational problems back to the application following a call by setting the "all" field of the **status_$t** structure. A value of **status_$ok** indicates that no errors were detected. Any other value implies that a problem occurred. The **status_$t** structure and the `error_$` routines can be used to display a textual representation of the error condition.

8.1 Data Types

This section describes the data types used in `error_$` routines.

The `error_$` routines take as input a status code in **status_$t** format.

status_$t

A status code. Most of the DECrpc routines supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
   long all;
}
```

However, the routines can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
 struct {
     unsigned fail : 1,
            subsys : 7,
            modc : 8;
     short    code;
 } s;
 long all;
} status_u;
```

all

All 32 bits in the status code. If **all** is equal to **status_$ok**, the routine that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

modc This indicates the module that encountered the error.

code This is a signed number that identifies the type of error that occurred.

Name

error_$c_get_text – return subsystem, module, and error texts for a status code

Format

void error_$c_get_text(*status, subsys, subsysmax, module, modulemax, error, errormax*)

status_$t *status*;
char **subsys*;
long *subsysmax*;
char **module*;
long *modulemax*;
char **error*;
long *errormax*;

Arguments

status	A status code in **status_$t** format.
subsys	A character string. The subsystem represented by the status code.
subsysmax	The maximum number of bytes to be returned in *subsys*.
module	A character string. The module represented by the status code.
modulemax	The maximum number of bytes to be returned in *module*.
error	A character string. The error represented by the status code.
errormax	The maximum number of bytes to be returned in *error*.

Description

The `error_$c_get_text` routine returns predefined text strings that describe the subsystem, the module, and the error represented by a status code. The strings are null terminated.

Data Types

This section describes the data types used in this `error_$` routine.

The `error_$` routine take as input a status code in **status_$t** format.

status_$t

A status code. Most of the DECrpc routines supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
   long all;
}
```

However, the routines can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
 struct {
     unsigned fail : 1,
            subsys : 7,
            modc : 8;
     short    code;
 } s;
 long all;
} status_u;
```

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the routine that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

modc This indicates the module that encountered the error.

code This is a signed number that identifies the type of error that occurred.

Files

```
RPC$LIB:RPC$STCODE.DAT
```

Name

error_$c_text – return an error message for a status code

Format

void error_$c_text(*status, message, messagemax***)**
status_$t *status***;**
char **message***;**
int *messagemax***;**

Arguments

status	A status code in **status_$t** format.
message	A character string. The error message represented by the status code.
messagemax	The maximum number of bytes to be returned in *message*.

Description

The `error_$c_text` routine returns a null terminated error message for reporting the completion status of a routine. The error message is composed from predefined text strings that describe the subsystem, the module, and the error represented by the status code.

Files

`RPC$LIB:RPC$STCODE.DAT`

lb_$ Routine Reference Pages 9

This chapter contains reference pages for the `lb_$` routines, which implement the programmatic interface to the Location Broker Client Agent.

The **lb_$** interface is defined by these files:

On VMS systems	RPC$IDL:LB.IDL
On ULTRIX systems	`/usr/include/idl/c/glb.h`

9.1 External Variables

This section describes the external variable used in `lb_$` routines.

uuid_$nil — An external **uuid_$t** variable that is preassigned the value of the nil UUID. Do not change the value of this variable.

9.2 Constants

This section describes constants used in `lb_$` routines.

lb_$default_lookup_handle
: Used as an input in Location Broker lookup routines. Specifies that a lookup is to start searching at the beginning of the database.

lb_$server_flag_local
: Used in the **flags** field of an **lb_$entry_t** variable. Specifies that an entry is to be registered only in the Local Location Broker (LLB) database. See the description of **lb_$server_flag_t** in Section 9.3.

status_$ok — A constant used to check status. If a completion status is equal to **status_$ok**, then the routine that supplied it was successful.

9.3 Data Types

This section describes data types used in `lb_$` routines.

lb_$entry_t — An identifier for an object, a type, an interface, and the socket address used to access a server exporting the interface to the object. The **lb_$entry_t** type is defined as follows:

```
typedef struct lb_$entry_t lb_$entry_t;
struct lb_$entry_t {
    uuid_$t object;
    uuid_$t obj_type;
    uuid_$t obj_interface;
    lb_$server_flag_t flags;
    ndr_$char annotation[64];
    ndr_$ulong_int saddr_len;
    socket_$addr_t saddr;
};
```

object A **uuid_$t**. The UUID for the object. Can be **uuid_$nil** if no object is associated.

obj_type A **uuid_$t**. The UUID for the type of the object. Can be **uuid_$nil** if no type is associated.

obj_interface A **uuid_$t**. The UUID for the interface. Can be **uuid_$nil** if no interface is associated.

flags An **lb_$server_flag_t**. Must be 0 or **lb_$server_flag_local**. A value of 0 specifies that the entry is to be registered in both the Local Location Broker (LLB) and global Location Broker (GLB) databases. A value of **lb_$server_flag_local** specifies registration only in the LLB database.

annotation A 64-character array. User-defined textual annotation.

saddr_len A 32-bit integer. The length of the **saddr** field.

saddr A **socket_$addr_t**. The socket address of the server.

lb_$lookup_handle_t

A 32-bit integer used to specify the location in the database at which a Location Broker lookup operation will start.

lb_$server_flag_t

A 32-bit integer used to specify the Location Broker databases in which an entry is to be registered. A value of 0 specifies registration in both the Local Location Broker (LLB) and Global Location Broker (GLB) databases. A value of **lb_$server_flag_local** specifies registration only in the LLB database.

socket_$addr_t

A socket address record that uniquely identifies a socket.

status_$t

A status code. Most of the DECrpc routines supply a completion code in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the system calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all — All 32 bits in the status code. If **all** is equal to **status_$ok**, the system call that supplied the status was successful.

fail — If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys — This indicates the subsystem that encountered the error.

modc — This indicates the module that encountered the error.

code — This is a signed number that identifies the type of error that occurred.

uuid_$t

A 128-bit value that uniquely identifies an object, type, or interface for all time.

9.4 Example

The following statement looks up information in the GLB database about a matrix multiplication interface:

```
lb_$lookup_interface (&matrix_id, &lookup_handle, max_results,
    &num_results, &matrix_results, &st);
```

lb_$lookup_interface

Name

lb_$lookup_interface – look up information about an interface in the Global Location Broker database

Format

#include <lb.h>

void lb_$lookup_interface(*obj_interface, lookup_handle, max_num_results, num_results, results, status*)

uuid_$t **obj_interface*;
lb_$lookup_handle_t **lookup_handle*;
unsigned long *max_num_results*;
unsigned long * *num_results*;
lb_$entry_t *results*[ ];
*status_$t *status;*

Arguments

obj_interface — The UUID of the interface being looked up.

lookup_handle — A location in the database.

On input, the *lookup_handle* indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database.

On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database. Any other return value indicates that the search found, at most, *max_num_results* matching entries before it reached the end of the database.

max_num_results — The maximum number of entries that can be returned by a single routine. This should be the number of elements in the *results* array.

num_results — The number of entries that were returned in the *results* array.

results	An array that contains the matching GLB database entries, up to the number specified by the *max_num_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.
status	The completion status.

Description

The `lb_$lookup_interface` routine returns GLB database entries whose *obj_interface* fields match the specified interface. It returns information about objects that can be accessed through that interface.

The `lb_$lookup_interface` routine cannot return more than *max_num_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup routines to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the locations of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup routine will skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Example

The following statement looks up information in the GLB database about a matrix multiplication interface:

```
lb_$lookup_interface (&matrix_id, &lookup_handle, max_results,
                        &num_results, &matrix_results, &st);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine.

lb_$database_invalid The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered The Location Broker does not have any entries that match the criteria specified in the lookup or unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database. If you are using an `lb_$lookup_object_local` or `lb_$lookup_range` routine specifying an LLB, check that you have specified the correct LLB.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

1. The database does not exist.

2. The database exists, but the Location Broker cannot access it.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested GLB or LLB. A communications failure occurred or the broker was not running.

Files

`RPC$INCLUDE:/glb.h`

See Also

intro, lb_$lookup_object, lb_$lookup_range, lb_$lookup_type

Name

lb_$lookup_object – look up information about an object in the Global Location Broker database

Format

#include <lb.h>

void lb_$lookup_object(*object, lookup_handle, max_num_results, num_results, results, status)*
uuid_$t **object*;
lb_$lookup_handle_t **lookup_handle*;
unsigned long *max_num_results*;
unsigned long * *num_results*;
lb_$entry_t *results[ ]*;
status_$t **status*;

Arguments

object	The UUID of the object being looked up.
lookup_handle	A location in the database. On input, the *lookup_handle* indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database. On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database. Any other return value indicates that the search found, at most, *max_num_results* matching entries before it reached the end of the database.
max_num_results	The maximum number of entries that can be returned by a single routine. This should be the number of elements in the *results* array.
num_results	The number of entries that were returned in the *results* array.

results — An array that contains the matching GLB database entries, up to the number specified by the *max_num_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.

status — The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `lb_$lookup_object` routine returns GLB database entries whose *object* field matches the specified object UUID.

The `lb_$lookup_object` routine cannot return more than *max_num_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup routines to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the locations of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup routine will skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Example

The following statement, looks up GLB database entries for the object identified by `bank_id`:

```
lb_$lookup_object(&bank_id, &lookup_handle,
                  MAX_LOCS, &n_locs, bank_loc, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid — The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered The Location Broker does not have any entries that match the criteria specified in the lookup or unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database. If you are using an `lb_$lookup_object_local` or `lb_$lookup_range` routine specifying an LLB, check that you have specified the correct LLB.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

- The database does not exist.
- The database exists, but the Location Broker cannot access it.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested GLB or LLB. A communications failure occurred or the broker was not running.

Files

`RPC$IDL:LB.IDL`

See Also

lb_$lookup_interface, lb_$lookup_object_local, lb_$lookup_range, lb_$lookup_type

lb_$lookup_object_local

Name

lb_$lookup_object_local – look up information about an object in a Local Location Broker database

Format

#include <lb.h>

void lb_$lookup_object_local(*object, location, location_length, lookup_handle max_num_results, num_results, results, status)*
uuid_$t **object;*
socket_$addr_t **location;*
unsigned long *location_length;*
lb_$lookup_handle_t **lookup_handle;*
unsigned long *max_num_results;*
unsigned long **num_results;*
lb_$entry_t *results[ ];*
status_$t **status;*

Arguments

object — The UUID of the object being looked up.

location — The location of the LLB database to be searched. The socket address must specify the network address of a host. However, the port number in the socket address is ignored, and the lookup request is always sent to the LLB port.

location_length — The length, in bytes, of the socket address specified by the location field.

lookup_handle — A location in the database.

On input, the *lookup_handle* indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database.

On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database. Any other return value

	indicates that the search found, at most, *max_num_results* matching entries before it reached the end of the database.
max_num_results	The maximum number of entries that can be returned by a single routine. This should be the number of elements in the *results* array.
num_results	The number of entries that were returned in the *results* array.
results	An array that contains the matching GLB database entries, up to the number specified by the *max_num_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `lb_$lookup_object_local` routine searches the specified LLB database and returns all entries whose *object* field matches the specified object.

The `lb_$lookup_object_local` routine cannot return more than *max_num_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup routines to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the locations of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup routine will skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Example

The following statement looks up information about the object **locobj**. Because there is only one entry on any host, the routine will return at most one result:

```
lb_$lookup_object_local (&locobj_id, &location, location_length,
                         &lookup_handle, 1, &num_results,
                         &results, &status);
```

Diagnostics

This section lists status codes for errors returned by this lb_$ routine in status.all.

lb_$database_invalid The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered The Location Broker does not have any entries that match the criteria specified in the lookup or unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database. If you are using an lb_$lookup_object_local or lb_$lookup_range routine specifying an LLB, check that you have specified the correct LLB.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

- The database does not exist.
- The database exists, but the Location Broker cannot access it.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested LLB. A communications failure occurred or the broker was not running.

Files

RPC$IDL:LB.IDL

See Also

lb_$lookup_range

Name

lb_$lookup_range – look up information in a Global Location Broker or Local Location Broker database

Format

#include <lb.h>

void lb_$lookup_range(*object, obj_type, obj_interface, location, location_length, lookup_handle, max_num_results, num_results, results, status)*
uuid_$t **object*;
uuid_$t **obj_type*;
uuid_$t **obj_interface*;
socket_$addr_t **location*;
unsigned long *location_length*;
lb_$lookup_handle_t **lookup_handle*;
unsigned long *max_num_results*;
unsigned long **num_results*;
lb_$entry_t *results[ ]*;
status_$t **status)*;

Arguments

object — The UUID of the object being looked up.

obj_type — The UUID of the type being looked up.

obj_interface — The UUID of the interface being looked up.

location — The location of the database to be searched. If the value of *location_length* is 0, the GLB database is searched. Otherwise, the LLB database at the host specified by *location* is searched; in this case, the port number in the socket address is ignored, and the lookup request is sent to the LLB port.

location_length — The length, in bytes, of the socket address specified by the *location* field. A value of 0 indicates that the GLB database is to be searched.

lookup_handle — A location in the database.

On input, the *lookup_handle* indicates the location in the database where the search begins. An input value of

lb_$default_lookup_handle specifies that the search will start at the beginning of the database.

On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database. Any other return value indicates that the search found, at most, *max_num_results* matching entries before it reached the end of the database.

max_num_results
: The maximum number of entries that can be returned by a single routine. This should be the number of elements in the *results* array.

num_results
: The number of entries that were returned in the *results* array.

results
: An array that contains the matching GLB database entries, up to the number specified by the *max_num_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.

status
: The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `lb_$lookup_range` routine returns database entries whose *object, obj_type,* and *obj_interface* fields match the specified values. A value of **uuid_$nil** in any of these input parameters acts as a wildcard and will match any value in the corresponding entry field. You can specify wildcards in any combination of these parameters.

The `lb_$lookup_range` routine cannot return more than *max_num_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup routines to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the locations of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup routine will skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Example

The following statement looks up information in the GLB database about servers that export the **matrix** interface for any objects of type **array.** The variable **glb** is defined elsewhere as a null pointer.

```
lb_$lookup_range(&uuid_$nil, &array_id, &matrix_id, glb, 0,
                         &lookup_handle, max_results,
                         &num_results, results, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered The Location Broker does not have any entries that match the criteria specified in the lookup or unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database. If you are using an `lb_$lookup_object_local` or `lb_$lookup_range` routine specifying an LLB, check that you have specified the correct LLB.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

- The database does not exist.
- The database exists, but the Location Broker cannot access it.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested LLB. A communications failure occurred or the broker was not running.

lb_$lookup_range

Files

RPC$IDL:LB.IDL

See Also

lb_$lookup_interface, lb_$lookup_object, lb_$lookup_object_local, lb_$lookup_type

Name

lb_$lookup_type – look up information about a type in the Global Location Broker database

Format

#include <lb.h>

void lb_$lookup_type(*obj_type, lookup_handle, max_num_results, num_results, results, status)*
uuid_$t **obj_type;*
lb_$lookup_handle_t **lookup_handle;*
unsigned long *max_num_results;*
unsigned long **num_results;*
lb_$entry_t *results*[ ];
status_$t **status*;

Arguments

obj_type — The UUID of the type being looked up.

lookup_handle — A location in the database.

On input, the *lookup_handle* indicates the location in the database where the search begins. An input value of **lb_$default_lookup_handle** specifies that the search will start at the beginning of the database.

On return, the *lookup_handle* indicates the next unsearched part of the database (that is, the point at which the next search should begin). A return value of **lb_$default_lookup_handle** indicates that the search reached the end of the database. Any other return value indicates that the search found, at most, *max_num_results* matching entries before it reached the end of the database.

max_num_results — The maximum number of entries that can be returned by a single routine. This should be the number of elements in the *results* array.

num_results — The number of entries that were returned in the *results* array.

results — An array that contains the matching GLB database entries, up to the number specified by the *max_num_results* parameter. If the array contains any entries for servers on the local network, those entries appear first.

status — The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `lb_$lookup_type` routine returns GLB database entries whose *obj_type* fields match the specified type. It returns information about all objects of that type and about all interfaces to each of these objects.

The `lb_$lookup_type` routine cannot return more than *max_num_results* matching entries at a time. The *lookup_handle* parameter enables you to find all matching entries by doing sequential lookups.

If you use a sequence of lookup routines to find entries in the database, it is possible that the returned results will skip or duplicate entries. This is because the Location Broker does not prevent modification of the database between lookups, and such modification can change the locations of entries relative to a *lookup_handle* value.

It is also possible that the results of a single lookup routine will skip or duplicate entries. This can occur if the size of the results exceeds the size of an RPC packet (64K bytes).

Example

The following statement looks up information in the GLB database about the type **array** :

```
lb_$lookup_type (&array_id, &lookup_handle, max_results,
                      &num_results, &results, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid — The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered The Location Broker does not have any entries that match the criteria specified in the lookup or unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database. If you are using an `lb_$lookup_object_local` or `lb_$lookup_range` routine specifying an LLB, check that you have specified the correct LLB.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

- The database does not exist, and the Location Broker cannot create it.
- The database exists, but the Location Broker cannot access it.
- The GLB entry table is full.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested GLB or LLB. A communications failure occurred or the broker was not running.

Files

`RPC$IDL:LB.IDL`

See Also

lb_$lookup_interface, lb_$lookup_object, lb_$lookup_range

lb_$register

Name

lb_$register – register an object and an interface with the Location Broker

Format

#include <lb.h>

void lb_$register(*object, obj_type, obj_interface, flags, annotation[64], location, location_length, entry, status*)
uuid_$t **object*;
uuid_$t **obj_type*;
uuid_$t **obj_interface*;
lb_$server_flag_t *flags*;
unsigned char *annotation*[**64**];
socket_$addr_t **location*;
unsigned long *location_length*;
lb_$entry_t **entry*;
status_$t **status*;

Arguments

object	The UUID of the object being registered.
obj_type	The UUID of the type of the object being registered.
obj_interface	The UUID of the interface being registered.
flags	Must be either **lb_$server_flag_local** (specifying registration with only the LLB at the local host) or 0 (specifying registration with both the LLB and the GLB).
annotation	A character array used only for informational purposes. This field can contain a textual description of the object and the interface. For proper display by the `lb_$admin` tool, the *annotation* should be terminated by a null character.
location	The socket address of the server that exports the interface to the object.
location_length	The length, in bytes, of the socket address specified by the *location* field.
entry	A copy of the entry that was entered in the Location Broker database.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `lb_$register` routine registers with the Location Broker an interface to an object and the location of a server that exports that interface. This routine replaces any existing entry in the Location Broker database that matches *object, obj_type, obj_interface,* and both the address family and host in *location;* if no such entry exists, the routine adds a new entry to the database.

If the *flags* parameter is `lb_$server_flag_local`, the entry is registered only in the LLB database at the host where the call is issued. Otherwise, the flag should be 0 to register with both the LLB and the GLB databases.

Example

The following statement registers the `bank` interface to the object identified by `bank_id`:

```
lb_$register (&bank_id, &bank_$uuid, &bank_$if_spec.id, 0,
                        BankName, &saddr, slen, &entry, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid The format of the Location Broker database is out of date. The database may have been created by an old version of the Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy The Location Broker database is currently in use in an incompatible manner.

lb_$update_failed The Location Broker was unable to register the entry.

lb_$cant_access The Location Broker cannot access the database. Among the possible reasons:

- The database does not exist, and the Location Broker cannot create it.

- The database exists, but the Location Broker cannot access it.
- The GLB entry table is full.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested GLB or LLB. A communications failure occurred or the broker was not running.

Files

RPC$IDL:LB.IDL

See Also

lb_$unregister

Name

lb_$unregister – remove an entry from the Location Broker database

Format

#include <lb.h>

void lb_$unregister(*entry, status***)**
lb_$entry_t ****entry*;**
status_$t ****status*;**

Arguments

entry The entry being removed from the Location Broker database.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `lb_$unregister` routine removes from the Location Broker database the entry that matches *entry*. The value of *entry* should be identical to that returned by the `lb_$register` routine when the database entry was created. However, `lb_$unregister` does not compare all of the fields in *entry*, the **annotation** field, and the port number in the **saddr** field.

This routine removes the entry from the LLB database on the local host (the host that issues the routine). If the **flags** field of *entry* is equal to 0, it removes the entry from the GLB database. If the **flags** field is equal to **lb_$server_flag_local,** it deletes only the LLB entry.

Example

The following statement unregisters the entry specified by `BankEntry,` which was obtained from a previous `lb_$register` routine:

```
lb_$unregister (&BankEntry, &status);
```

Diagnostics

This section lists status codes for errors returned by this `lb_$` routine in `status.all`.

lb_$database_invalid The format of the Location Broker database is out of date. The database may have been created by an old version of the

Location Broker; in this case, delete the out-of-date database and reregister any entries that it contained. The LLB or GLB that was accessed may be running out-of-date software; in this case, update all Location Brokers to the current software version.

lb_$database_busy
The Location Broker database is currently in use in an incompatible manner.

lb_$not_registered
The Location Broker does not have any entries that match the criteria specified in the unregister routine. The requested object, type, interface, or combination thereof is not registered in the specified database.

lb_$update_failed
The Location Broker was unable to register or unregister the entry.

lb_$cant_access
The Location Broker cannot access the database. Among the possible reasons:

- The database does not exist.
- The database exists, but the Location Broker cannot access it.

lb_$server_unavailable
The Location Broker Client Agent cannot reach the requested GLB or LLB. A communications failure occurred or the broker was not running.

Files

```
RPC$IDL:LB.IDL
```

See Also

lb_$register

pfm_$ Routine Reference Pages 10

This chapter contains reference pages for the `pfm_$` routines, which allow programs to manage signals, faults, and exceptions by establishing clean-up handlers.

A clean-up handler is a piece of code that ensures a program terminates gracefully when it receives a fatal error. A clean-up handler begins with a `pfm_$cleanup` call, and usually ends with a call to `pfm_$signal` or `pgm_$exit`, though it can also simply continue back into the program after the clean-up code.

A clean-up handler is not entered until all fault handlers established for a fault have returned. If there is more than one established clean-up handler for a program, the most recently established clean-up handler is entered first, followed by the next most recently established clean-up handler, and so on to the first established clean-up handler if necessary.

There is a default clean-up handler invoked after all user-defined handlers have completed. It releases any resources still held by the program, before returning control to the process that invoked it.

10.1 Constants

This section describes the constants used by the `pfm_$` routines.

pfm_$init_signal_handlers
: A constant used as the *flags* parameter to `pfm_$init`, causing C signals to be intercepted and converted to PFM signals.

10.2 Data Types

This section describes the data types used in `pfm_$` routines.

pfm_$cleanup_rec
: A record type for passing process context among clean-up handler routines. It is an opaque data type.

status_$t
: A status code. Most of the DECrpc routines supply a completion code in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the system calls can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the system call that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

modc This indicates the module that encountered the error.

code This is a signed number that identifies the type of error that occurred.

Name

pfm_$cleanup – establish a cleanup handler

Format

#include <base.h>
#include <pfm.h>

status_$t pfm_$cleanup (*cleanup_record*)
pfm_$cleanup_rec **cleanup_record;*

Arguments

cleanup_record — A record of the context when `pfm_$cleanup` is called. A program should treat this as an opaque data structure and not try to alter or copy its contents. It is needed by `pfm_$rls_cleanup` and `pfm_$reset_cleanup` to restore the context of the calling process at the cleanup handler entry point.

Description

The `pfm_$cleanup` routine establishes a cleanup handler that is executed when a fault occurs. A cleanup handler is a piece of code executed before a program exits when a signal is received by the process. The cleanup handler begins where `pfm_$cleanup` is called; the `pfm_$cleanup` routine registers an entry point with the system where program execution resumes when a fault occurs. When a fault occurs, execution resumes after the most recent call to `pfm_$cleanup.`

There can be more than one cleanup handler in a program. Multiple cleanup handlers are executed consecutively on a last-in/first-out basis, starting with the most recently established handler and ending with the first cleanup handler. The system provides a default cleanup handler established at program invocation. The default cleanup handler is always called last, just before a program exits, and releases any system resources still held, before returning control to the process that invoked the program.

Diagnostics

When called to establish a cleanup handler, `pfm_$cleanup` returns the status **pfm_$cleanup_set** to indicate the cleanup handler was successfully established. When the cleanup handler is entered in response to a fault signal, `pfm_$cleanup` effectively returns the value of the fault that triggered the handler.

pfm_$cleanup

This section lists status codes for errors returned by this `pfm_$` routine in `status.all`.

pfm_$bad_rls_order Attempted to release a cleanup handler out of order.

pfm_$cleanup_not_found
There is no pending cleanup handler.

pfm_$cleanup_set A cleanup handler was established successfully.

pfm_$cleanup_set_signalled
Attempted to use **pfm_$cleanup_set** as a signal.

pfm_$invalid_cleanup_rec
Passed an invalid cleanup record to a routine.

pfm_$no_space Cannot allocate storage for a cleanup handler.

NOTE

Clean-up handler code runs with asynchronous faults inhibited.

When `pfm_$cleanup` returns something other than **pfm_$cleanup_set** indicating that a fault has occurred, there are four possible ways to leave the cleanup code:

- The program can call `pfm_$signal` to start the next cleanup handler with a different fault signal.
- The program can call `pgm_$exit` to start the next cleanup handler with the same fault signal.
- The program can continue with the code following the cleanup handler. It should generally call `pfm_$enable` to reenable asynchronous faults. Execution continues from the end of the cleanup handler code; it does not resume where the fault signal was received.
- The program can reestablish the handler by calling `pfm_$reset_cleanup` before proceeding.

Files

RPC$INCLUDE:BASE.H
RPC$INCLUDE:PFM.H

See Also

pfm_$signal

Name

pfm_$enable – enable asynchronous faults

Format

#include <base.h>
#include <pfm.h>

void pfm_$enable()

Description

The `pfm_$enable` routine enables asynchronous faults after they have been inhibited by a routine to `pfm_$inhibit;` `pfm_$enable` causes the operating system to pass asynchronous faults to the calling process.

While faults are inhibited, the operating system holds, at most, one asynchronous fault. Consequently, when `pfm_$enable` returns, there can be, at most, one fault waiting on the process. If more than one fault was received between routines to `pfm_$inhibit` and `pfm_$enable`, the process receives the first asynchronous fault received while faults were inhibited.

Files

RPC$INCLUDE:BASE.H
RPC$INCLUDE:PFM.H

See Also

pfm_$enable_faults, pfm_$inhibit

pfm_$enable_faults

Name

pfm_$enable_faults – enable asynchronous faults

Format

#include <base.h>
#include <pfm.h>

void pfm_$enable_faults()

Description

The `pfm_$enable_faults` routine enables asynchronous faults after they have been inhibited by a call to `pfm_$inhibit_faults; pfm_$enable_faults` causes the operating system to pass asynchronous faults on to the calling process.

While faults are inhibited, the operating system holds, at most, one asynchronous fault. Consequently, when `pfm_$enable_faults` returns, there can be, at most, one fault waiting on the process. If more than one fault was received between routines to `pfm_$inhibit_faults` and `pfm_$enable_faults,` the process receives the first asynchronous fault received while faults were inhibited.

Diagnostics

This section lists the status codes for errors returned by this `pfm_$` routine.

pfm_$bad_rls_order Attempted to release a cleanup handler out of order.

pfm_$cleanup_not_found
There is no pending cleanup handler.

pfm_$cleanup_set A cleanup handler was established successfully.

pfm_$cleanup_set_signalled
Attempted to use **pfm_$cleanup_set** as a signal.

pfm_$invalid_cleanup_rec
Passed an invalid cleanup record to a routine.

pfm_$no_space Cannot allocate storage for a cleanup handler.

Files

RPC$INCLUDE:BASE.H
RPC$INCLUDE:PFM.H

See Also

pfm_$enable, pfm_$inhibit_faults

pfm_$inhibit

Name

pfm_$inhibit – inhibit asynchronous faults

Format

#include <base.h>
#include <pfm.h>

void pfm_$inhibit()

Description

The `pfm_$inhibit` routine prevents asynchronous faults from being passed to the calling process. While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, a call to `pfm_$inhibit` can result in the loss of some signals. It is good practice to inhibit faults only when absolutely necessary.

NOTE

This routine has no effect on the processing of synchronous faults, such as floating-point and overflow exceptions, access violations, and so on.

Files

RPC$INCLUDE:BASE.H
RPC$INCLUDE:PFM.H

See Also

pfm_$enable, pfm_$inhibit_faults

Name

pfm_$inhibit_faults – inhibit asynchronous faults

Format

#include <base.h>
#include <pfm.h>

void pfm_$inhibit_faults()

Description

The `pfm_$inhibit_faults` routine prevents asynchronous faults from being passed to the calling process. While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, a call to `pfm_$inhibit_faults` can result in the loss of some signals. It is good practice to inhibit faults only when absolutely necessary.

NOTE

This call has no effect on the processing of synchronous faults such as floating-point and overflow exceptions, access violations, and so on.

Files

RPC$INCLUDE:BASE.H
RPC$INCLUDE:PFM.H

See Also

pfm_$enable_faults, pfm_$inhibit

pfm_$init

Name

pfm_$init – initialize the PFM package

Format

#include <base.h>
#include <pfm.h>

void pfm_$init(*flags*)
unsigned long *flags;*

Arguments

flags

pfm_$init_signal_handlers
Currently the only valid flag value. A flag's variable must be set to contain this value or the call will perform no initialization. A call to **pfm_$init_signal_handlers** causes C signals to be intercepted and converted to PFM signals. On ULTRIX and VMS systems, the signals intercepted are SIGINIT, SIGILL, SIGFPE, SIGTERM, SIGHUP, SIGQUIT, SIGTRAP, SIGBUS, SIGSEGV, and SIGSYS.

Description

The call to `pfm_$init()` establishes a default set of signal handlers for the routine. The call to `pfm_$init()` should be made prior to the application's use of all other runtime RPC routines. This enables the RPC runtime system to catch and report all fault and/or interrupt signals that may occur during normal operation. Additionally, the user may provide a fault processing cleanup handler for application-specific exit handling.

Files

RPC$INCLUDE:BASE.H
RPC$INCLUDE:PFM.H

See Also

pfm_$cleanup

Name

pfm_$reset_cleanup – reset a cleanup handler

Format

#include <base.h>
#include <pfm.h>

void pfm_$reset_cleanup(*&cleanup_record, &status***)**
pfm_$cleanup_rec **cleanup_record*;
status_$t **status*;

Arguments

cleanup_record — A record of the context at the cleanup handler entry point. It is supplied by `pfm_$cleanup`, when the cleanup handler if first established.

status — The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `pfm_$reset_cleanup` routine reestablishes the cleanup handler last entered, so that any subsequent errors enter it first. Use this procedure only within cleanup handler code.

Diagnostics

This section lists status codes for errors returned by this `pfm_$` routine in `status.all`.

pfm_$bad_rls_order — Attempted to release a cleanup handler out of order.

pfm_$cleanup_not_found — There is no pending cleanup handler.

pfm_$cleanup_set — A cleanup handler was established successfully.

pfm_$invalid_cleanup_rec — Passed an invalid cleanup record to a routine.

pfm_$no_space — Cannot allocate storage for a cleanup handler.

pfm_$reset_cleanup

Files

RPC$INCLUDE:BASE.H
RPC$INCLUDE:PFM.H

Name

pfm_$rls_cleanup – release cleanup handlers

Format

#include <base.h>
#include <pfm.h>

void pfm_$rls_cleanup(*&cleanup_record, &status)*
pfm_$cleanup_rec **cleanup_record*;
status_$t **status*;

Arguments

cleanup_record — The cleanup record for the first cleanup handler to release.

status — The completion status. If *status* is **pfm_$bad_rls_order**, it means that the caller attempted to release a cleanup handler before releasing all handlers established after it. This status is only a warning; the intended cleanup handler is released, along with all cleanup handlers established after it. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `pfm_$rls_cleanup` routine releases the cleanup handler associated with *cleanup_record* and all cleanup handlers established after it.

Diagnostics

This section lists the status codes for errors returned by this `pfm_$` routine in `status.all`.

pfm_$bad_rls_order — Attempted to release a cleanup handler out of order.

pfm_$cleanup_not_found — There is no pending cleanup handler.

pfm_$cleanup_set — A cleanup handler was established successfully.

pfm_$cleanup_set_signalled — Attempted to use **pfm_$cleanup_set** as a signal.

pfm_$rls_cleanup

pfm_$invalid_cleanup_rec
Passed an invalid cleanup record to a routine.

Files

RPC$INCLUDE:BASE.H
RPC$INCLUDE:PFM.H

Name

pfm_$signal – signal the calling process

Format

#include <base.h>
#include <pfm.h>

void pfm_$signal(*fault_signal***)**
status_$t **fault_signal*;

Arguments

fault_signal	A fault code.

Description

The `pfm_$signal` routine signals the fault specified by *fault_signal* to the calling process. It is usually called to leave cleanup handlers.

NOTE

This routine does not return when successful.

Diagnostics

This section lists status codes for errors returned by this `pfm_$` routine.

pfm_$bad_rls_order Attempted to release a cleanup handler out of order.

pfm_$cleanup_not_found
There is no pending cleanup handler.

pfm_$cleanup_set A cleanup handler was established successfully.

pfm_$cleanup_set_signalled
Attempted to use **pfm_$cleanup_set** as a signal.

pfm_$invalid_cleanup_rec
Passed an invalid cleanup record to a routine.

pfm_$no_space Cannot allocate storage for a cleanup handler.

Files

RPC$INCLUDE:BASE.H
RPC$INCLUDE:PFM.H

pgm_$ Routine Reference Pages 11

This chapter contains the reference page for the `pgm_$init` command.

pgm_$exit

Name

pgm_$exit – exit a program

Format

#include <base.h>
#include <pfm.h>

void pgm_$exit

Description

The `pgm_$exit` routine exits from the calling program and returns control to the process that invoked it. When `pgm_$exit` is called, any files left open by the program are closed, any storage acquired is released, and asynchronous faults are reenabled if they were inhibited by the calling program.

The `pgm_$exit` routine always calls `pfm_$signal` with a status of **status_$ok.**

Files

RPC$INCLUDE:BASE.H
RPC$INCLUDE:PFM.H

rpc_$ Routine Reference Pages 12

This chapter contains reference pages for the `rpc_$` routines, which implement the DECrpc remote procedure call (RPC) mechanism. The `rpc_$` interface is defined by these files:

On VMS systems	RPC$IDL:RPC.IDL
On ULTRIX systems	`/usr/include/idl/rpc.idl`

Most of the `rpc_$` routines can be used only by clients or only by servers. This aspect of their usage is specified at the beginning of each routine description, in the Name section.

12.1 External Variables

This section describes the external variable used in `rpc_$` routines.

uuid_$nil
An external **uuid_$t** variable that is preassigned the value of the nil UUID. Do not change the value of this variable.

12.2 Constants

This section describes constants used in `rpc_$` routines.

rpc_$mod
A module code indicating the RPC module.

status_$ok
A constant used to check status. If a completion status is equal to **status_$ok**, then the routine that supplied it was successful. See the description of the **status_$t** type.

rpc_$unbound_port
A port number indicating to the RPC run-time library that no port is specified. Identical to **socket_$unspec_port.**

The following 16-bit-integer constants are used to specify the communications protocol address families in **socket_$addr_t** structures. Note that several of the **rpc_$** and **socket_$** calls use the 32-bit-integer equivalents of these values.

socket_$unspec
Address family is unspecified.

socket_$internet
Internet Protocols (IP).

12.3 Data Types

This section describes data types used in `rpc_$` routines.

handle_t An RPC handle.

rpc_$epv_t An entry point vector (EPV). An array of **rpc_$server_stub_t**, pointers to server stub procedures.

rpc_$generic_epv_t
An entry point vector (EPV). An array of **rpc_$generic_server_stub_t**, pointers to generic server stub procedures.

rpc_$if_spec_t
An RPC interface specifier. This opaque data type contains information about an interface, including its UUID, the current version number, any well known ports used by servers that export the interface, and the number of operations in the interface.

rpc_$mgr_epv_t
An entry point vector (EPV). An array of pointers to manager procedures.

rpc_$shut_check_fn_t
A pointer to a function. If a server supplies this function pointer to **rpc_$allow_remote_shutdown**, the function will be called when a remote shutdown request arrives, and if the function returns true, the shutdown is allowed. The following C definition for **rpc_$shut_check_fn_t** illustrates the prototype for this function:

```
typedef boolean (*rpc_$shut_check_fn_t) (
    handle_t h,
    status_$t *st)
```

The handle argument can be used to determine information about the remote caller.

socket_$addr_t
A socket address record that uniquely identifies a socket.

status_$t A status code. Most of the DECrpc routines supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the routines can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the routine that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

modc This indicates the module that encountered the error.

code This is a signed number that identifies the type of error that occurred.

uuid_$t A 128-bit value that uniquely identifies an object, type, or interface for all time.

rpc_$alloc_handle

Name

rpc_$alloc_handle – create an RPC handle (client only)

Format

#include <rpc.h>

handle_t rpc_$alloc_handle(*&object, family, &status)*
uuid_$t **object*;
unsigned long *family*;
status_$t **status*;

Arguments

object	The UUID of the object to be accessed. If there is no specific object, specify **uuid_$nil**.
family	The address family to use in communications to access the object. Currently, only **socket_$ internet** is supported.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$alloc_handle` routine creates an unbound RPC handle that identifies a particular object but not a particular server or host.

If a remote procedure call is made using the unbound handle, it will effect a broadcast to all Local Location Brokers (LLBs) on the local network. If the call's interface and the object identified by the handle are both registered with any LLB, that LLB forwards the request to the registering server. The client RPC runtime library returns the first response that it receives and binds the handle to the first responding server.

Example

The following statement allocates a handle that identifies the Acme company's payroll database object:

```
h = rpc_$alloc_handle (&acme_pay_id, socket_$internet, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$comm_failure The client was unable to get a response from the server.

rpc_$unk_if The requested interface is not known. It is not registered in the server, the version number of the registered interface is different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.

rpc_$cant_create_sock
The RPC runtime library was unable to create a socket.

rpc_$cant_bind_sock The RPC runtime library created a socket but was unable to bind it to a socket address.

rpc_$wrong_boot_time
The server boot time value maintained by the client does not correspond to the current server boot time. The server was probably rebooted while the client program was running.

rpc_$not_in_call An internal error.

rpc_$you_crashed This error can occur if a server has crashed and restarted. A client RPC runtime library sends the error to the server if the client makes a remote procedure call before the server crashes, then receives a response after the server restarts.

rpc_$proto_error An internal protocol error.

Files

`RPC$IDL:RPC.IDL`

See Also

rpc_$free_handle, rpc_$set_binding

rpc_$allow_remote_shutdown

Name

rpc_$allow_remote_shutdown – allow or disallow remote shutdown of a server (server only)

Format

#include <rpc.h>

void rpc_$allow_remote_shutdown(*allow*, *checkproc*, *status*)
unsigned long *allow*;
rpc_$shut_check_fn_t *checkproc*;
status_$t **status*;

Arguments

allow	A value indicating "false" if zero, "true" otherwise.
checkproc	A pointer to a Boolean function.
status	The completion status.

Description

The `rpc_$allow_remote_shutdown` call allows or disallows remote callers to shut down a server using `rrpc_$shutdown`.

By default, servers do not allow remote shutdown via `rrpc_$shutdown`. If a server calls `rpc_$allow_remote_shutdown` with `allow` true (not zero) and *checkproc* nil, then remote shutdown will be allowed. If `allow` is true and *checkproc* is not nil, then when a remote shutdown request arrives, the function denoted by *checkproc* is called and the shutdown is allowed if the function returns true. If `allow` is false (zero), remote shutdown is disallowed.

Diagnostics

This section lists status codes for errors returned by **rpc_$** calls.

rpc_$comm_failure	The client was unable to get a response from the server.
rpc_$op_rng_error	The requested operation does not correspond to a valid operation in the requested interface.
rpc_$unk_if	The requested interface is not known. It is not registered in the server, the version number of the registered interface is

different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.

rpc_$cant_create_sock
The RPC runtime library was unable to create a socket.

rpc_$cant_bind_sock The RPC runtime library created a socket but was unable to bind it to a socket address.

rpc_$wrong_boot_time
The server boot time value maintained by the client does not correspond to the current server boot time. The server was probably rebooted while the client program was running.

rpc_$too_many_ifs The maximum number of interfaces is already registered with the RPC runtime library; the server must unregister some interface before it registers an additional interface.

rpc_$not_in_call An internal error.

rpc_$you_crashed This error can occur if a server has crashed and restarted. A client RPC runtime library sends the error to the server if the client makes a remote procedure call before the server crashes, then receives a response after the server restarts.

rpc_$proto_error An internal protocol error.

rpc_$too_many_sockets
The server is trying to use more than the maximum number of sockets that is allowed; it has called **rpc_$use_family** or **rpc_$use_family_wk** too many times.

rpc_$illegal_register You are trying to register an interface that is already registered and you are using an EPV different from the one used when the interface was first registered. An interface can be multiply registered, but you must use the same EPV in each **rpc_$register** call.

rpc_$bad_pkt The server or client has received an ill-formed packet.

rpc_$unbound_handle
The handle is not bound and does not represent a particular host address. Returned by **rpc_$inq_binding**.

rpc_$addr_in_use The address and port specified in an **rpc_$use_family_wk** call are already in use. This is caused by multiple calls to **rpc_$use_family_wk** with the same well-known port.

rpc_$allow_remote_shutdown

Files

RPC$IDL:RPC.IDL

See Also

rpc_$shutdown, rrpc_$shutdown

Name

rpc_$bind – allocate an RPC handle and set its binding to a server (client only)

Format

#include <rpc.h>

handle_t rpc_$bind(*&object, &sockaddr, slength, &status)*
uuid_$t **object*;
socket_$addr_t **sockaddr*;
unsigned long *slength*;
status_$t **status*;

Arguments

object	The UUID of the object to be accessed. If there is no specific object, specify **uuid_$nil**.
sockaddr	The socket address of the server.
slength	The length, in bytes, of *sockaddr*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$bind` routine creates a fully bound RPC handle that identifies a particular object and server. This routine is equivalent to an `rpc_$alloc_handle` routine followed by an `rpc_$set_binding` routine.

Example

The following statement binds the binop client to the specified object and socket address. The **loc** parameter is the result of a previous call to `rpc_$name_to_sockaddr`, which converted the host name and port number to a socket address.

```
rh = rpc_$bind (&uuid_$nil, &loc, llen, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$cant_bind_sock The RPC runtime library created a socket but was unable to bind it to a socket address.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

Files

`RPC$IDL:RPC.IDL`

See Also

rpc_$clear_binding, rpc_$clear_server_binding, rpc_$set_binding

Name

rpc_$clear_binding – unset the binding of an RPC handle to a host and server (client only)

Format

#include <rpc.h>

void rpc_$clear_binding(*handle, status***)**
handle_t *handle***;**
status_$t **status***;**

Arguments

handle The RPC handle whose binding is being cleared.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$clear_binding` routine removes any association between an RPC handle and a particular server and host, but it does not remove the association between the handle and an object. This routine saves the RPC handle so that it can be reused to access the same object, either by broadcasting or after resetting the binding to another server.

A remote procedure call made using an unbound handle is broadcast to all Local Location Brokers (LLBs) on the local network. If the call's interface and the object identified by the handle are both registered with any LLB, that LLB forwards the request to the registering server. The client RPC runtime library returns the first response that it receives and binds the handle to the first server that responded.

The `rpc_$clear_binding` routine is the inverse of the `rpc_$set_binding` routine.

Example

Clear the binding represented in *handle*:

```
rpc_$clear_binding (handle, &status);
```

rpc_$clear_binding

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

Files

`RPC$IDL:RPC.IDL`

See Also

rpc_$bind, rpc_$clear_server_binding, rpc_$set_binding

Name

rpc_$clear_server_binding – unset the binding of an RPC handle to a server (client only)

Format

#include <rpc.h>

void rpc_$clear_server_binding(*handle, status***)**
handle_t *handle***;**
status_$t **status***;**

Arguments

handle	The RPC handle whose binding is being cleared.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$clear_server_binding` routine removes the association between an RPC handle and a particular server (that is, a particular port number), but does not remove the associations with an object and with a host (that is, a network address). This call replaces a fully bound handle with a bound-to-host handle. A bound-to-host handle identifies an object located on a particular host but does not identify a server exporting an interface to the object.

If a client uses a bound-to-host handle to make a remote procedure call, the call is sent to the Local Location Broker (LLB) forwarding port at the host identified by the handle. If the call's interface and the object identified by the handle are both registered with the host's LLB, the LLB forwards the request to the registering server. When the client RPC runtime library receives a response, it binds the handle to the server. Subsequent remote procedure calls that use this handle are then sent directly to the bound server's port.

The `rpc_$clear_server_binding` routine is useful for client error recovery when a server dies. The port that a server uses when it restarts is not necessarily the same port that it used previously; therefore, the binding that the client was using may not be correct. This routine enables the client to unbind from the dead server while retaining the binding to the host. When the client sends a request, the binding is automatically set to the server's new port.

rpc_$clear_server_binding

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

```
RPC$IDL:RPC.IDL
RPC$INCLUDE:RPC.H
```

See Also

rpc_$bind, rpc_$clear_binding, rpc_$set_binding

Name

rpc_$dup_handle – make a copy of an RPC handle (client only)

Format

#include <rpc.h>

handle_t rpc_$dup_handle(*handle, status***)**
handle_t *handle***;**
status_$t **status***;**

Arguments

handle The RPC handle to be copied.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$dup_handle` routine returns a copy of an existing RPC handle. Both handles can then be used in the client program for concurrent multiple accesses to a binding. Because all duplicates of a handle reference the same data, an `rpc_$set_binding`, `rpc_$clear_binding`, or `rpc_$clear_server_binding` routine made on any one duplicate affects all duplicates. However, an RPC handle is not freed until `rpc_$free_handle` is called on all copies of the handle.

Files

`RPC$IDL:RPC.IDL`

See Also

rpc_$alloc_handle, rpc_$free_handle

rpc_$free_handle

Name

rpc_$free_handle – free an RPC handle (client only)

Format

#include <rpc.h>

void rpc_$free_handle(*handle, status***)**
handle_t *handle***;**
status_$t **status***;**

Arguments

handle The RPC handle to be freed.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$free_handle` routine frees an RPC handle. This routine clears any association between the handle and a server or an object and releases the resources identified by the RPC handle. The client program cannot use a handle after it is freed.

Example

The following statement frees a handle:

```
rpc_$free_handle (handle, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

Files

`RPC$IDL:RPC.IDL`

See Also

rpc_$alloc_handle, rpc_$dup_handle

rpc_$inq_binding

Name

rpc_$inq_binding – return the socket address represented by an RPC handle (client or server)

Format

#include <rpc.h>

void rpc_$inq_binding(*handle, sockaddr, slength, status***)**
handle_t *handle***;**
socket_$addr_t **sockaddr***;**
unsigned long **slength***;**
status_$t **status***;**

Arguments

handle	An RPC handle.
sockaddr	The socket address represented by *handle.*
slength	The length, in bytes, of *sockaddr.*
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$inq_binding` routine enables a client to determine the socket address, and therefore the server, identified by an RPC handle. It is useful when a client uses an unbound handle in a remote procedure call and wants to determine the particular server that responded to the call.

Example

The Location Broker administrative tool, `lb_$admin`, uses the following statement to determine the GLB that last responded to a lookup request:

```
rpc_$inq_binding(lb_$handle, &global_broker_addr,
                    &global_broker_addr_len, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

rpc_$unbound_handle
The handle is not bound and does not represent a particular host address. Returned by `rpc_$inq_binding`.

Files

`RPC$IDL:RPC.IDL`

See Also

rpc_$bind, rpc_$set_binding

rpc_$inq_object

Name

rpc_$inq_object – return the object UUID represented by an RPC handle (client or server)

Format

#include <rpc.h>

void rpc_$inq_object(*handle, object, status***)**
handle_t *handle***;**
uuid_$t **object***;**
status_$t **status***;**

Arguments

handle	An RPC handle.
object	The UUID of the object identified by *handle*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$inq_object` routine enables a client or server to determine the particular object that a handle represents.

If a server exports an interface through which clients can access several objects, it can use `rpc_$inq_object` to determine the object requested in a call. This routine requires an RPC handle as input, so the server can make the call only if the interface uses explicit handles (that is, if each operation in the interface has a handle parameter). If the interface uses an implicit handle, the handle identifier is not passed to the server.

Example

A database server that manages multiple databases must determine the particular database to be accessed whenever it receives a remote procedure call. Each manager routine makes the following call; the routine then uses the returned UUID to identify the database to be accessed:

```
rpc_$inq_object (handle, &db_uuid, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$unk_if The requested interface is not known. It is not registered in the server, the version number of the registered interface is different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

Files

`RPC$IDL:RPC.IDL`

rpc_$listen

Name

rpc_$listen – listen for and handle remote procedure call (RPC) packets (server only)

Format

#include <rpc.h>

void rpc_$listen(*max_calls, status***)**
unsigned long *max_calls***;**
status_$t **status***;**

Arguments

max_calls — This value indicates the maximum number of calls that the server is allowed to process concurrently. On ULTRIX systems, the only recognized value is 1; any other value is ignored and defaulted to 1.

status — The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$listen` routine dispatches incoming remote procedure call requests to manager procedures and returns the responses to the client. You must issue `rpc_$use_family` or `rpc_$use_family_wk` before you use `rpc_$listen`. This routine normally does not return. A return from this routine indicates either an irrecoverable error, or that an `rpc_$shutdown` call has been issued. If `status.all` is equal to **status_$ok**, the assumption is that `rpc_$shutdown` has occurred.

Example

Listen for incoming remote procedure call requests.

```
rpc_$listen (1, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call	An internal error.
rpc_$you_crashed	This error can occur if a server has crashed and restarted. A client RPC runtime library sends the error to the server if the client makes a remote procedure call before the server crashes, then receives a response after the server restarts.
rpc_$proto_error	An internal protocol error.
rpc_$bad_pkt	The server or client has received an ill-formed packet.

Files

```
RPC$IDL:RPC.IDL
RPC$INCLUDE:RPC.H
```

See Also

rpc_$shutdown

rpc_$name_to_sockaddr

Name

rpc_$name_to_sockaddr – convert a host name and port number to a socket address (client or server)

Format

#include <rpc.h>

void rpc_$name_to_sockaddr(*name, nlength, port, family, &sockaddr, slength, status*)
unsigned char *name*;
unsigned long *nlength*;
unsigned long *port*;
unsigned long *family*;
socket_$addr_t **sockaddr*;
unsigned long **slength*;
status_$t **status*;

Arguments

name — A string that contains a host name and, optionally, a port and an address family. The format is *family*:*host*[*port*]. If you specify a *family* as part of the *name* parameter, you must specify **socket_$unspec** in the *family* parameter. The *family* part of the name parameter is **ip**; *host* is the host name; *port* is an integer port number.

nlength — The number of characters in *name*.

port — The socket port number. If you are not specifying a well-known port, give this parameter the value **rpc_$unbound_port**; in this case, the returned socket address will specify the Local Location Broker (LLB) forwarding port at *host*. If you specify the port number in the *name* parameter, this parameter is ignored.

family — The address family to use for the socket address. This value corresponds to the communications protocol used to access the socket and determines how the *sockaddr* is expressed. If you specify the address family in the *name* parameter, this parameter must have the value **socket_$unspec**.

sockaddr — The socket address corresponding to *name*, *port*, and *family*.

slength	The length, in bytes, of *sockaddr*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$name_to_sockaddr` routine provides the socket address for a socket, given the host name, the port number, and the address family.

You can specify the socket address information either as one text string in the *name* parameter or by passing each of the three elements as separate parameters(*name, port,* and *family*); in the latter case, use only the hostname in the *name* parameter.

NOTE

This routine has been superseded by the `socket_$from_name` routine.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

`RPC$IDL:RPC.IDL`

See Also

rpc_$sockaddr_to_name, socket_$from_name

rpc_$register

Name

rpc_$register – register an interface (server only)

Format

#include <rpc.h>

void rpc_$register(*ifspec, epv, status***)**
rpc_$if_spec_t **ifspec*;
rpc_$epv_t *epv*;
status_$t **status*;

Arguments

ifspec	The interface being registered.
epv	The entry point vector (EPV) for the operations in the interface. The EPV is always defined in the server stub that is generated by the NIDL compiler from an interface definition.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$register` routine registers an interface with the RPC runtime library. After an interface is registered, the RPC runtime library will pass requests for that interface to the server.

You can call `rpc_$register` several times with the same interface (e.g., from various subroutines of the same server), but each call must specify the same EPV. Each registration increments a reference count for the registered interface; an equal number of `rpc_$unregister` routines are then required to unregister the interface.

Example

The following statement registers the bank interface with the bank server host's RPC runtime library:

```
rpc_$register (&bank_$if_spec, bank_$server_epv, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$op_rng_error The requested operation does not correspond to a valid operation in the requested interface.

rpc_$too_many_ifs The maximum number of interfaces is already registered with the RPC runtime library; the server must unregister some interface before it registers an additional interface.

rpc_$not_in_call An internal error.

rpc_$you_crashed This error can occur if a server has crashed and restarted. A client RPC runtime library sends the error to the server if the client makes a remote procedure call before the server crashes, then receives a response after the server restarts.

rpc_$proto_error An internal protocol error.

rpc_$illegal_register You are trying to register an interface that is already registered and you are using an EPV different from the one used when the interface was first registered. An interface can be multiply registered, but you must use the same EPV in each `rpc_$register` routine.

rpc_$bad_pkt The server or client has received an ill-formed packet.

Files

`RPC$IDL:RPC.IDL`

See Also

rpc_$register_mgr, rpc_$register_object, rpc_$unregister

rpc_$register_mgr

Name

rpc_$register_mgr – register a manager (server only)

Format

#include <rpc.h>

void rpc_$register_mgr(*type, ifspec, sepv, mepv, status***)**
uuid_$t **type*;
rpc_$if_spec_t **ifspec*;
rpc_$generic_epv_t *sepv*;
rpc_$mgr_epv_t *mepv*;
status_$t **status*;

Arguments

type	The UUID of the type being registered.
ifspec	The interface being registered.
sepv	The generic EPV, a vector of pointers to server stub procedures.
mepv	The manager EPV, a vector of pointers to manager procedures.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$register_mgr` routine registers the set of manager procedures that implement a specified interface for a specified type.

Servers can invoke this routine several times with the same interface (*ifspec*) and generic EPV (*sepv*) but with a different object type (*type*) and manager EPV (*mepv*) on each invocation. This technique allows a server to export several implementations of the same interface.

Servers that export several versions of the same interface (but not different implementations for different types) must also use `rpc_$register_mgr`, not `rpc_$register`. Such servers should supply **uuid_$nil** as the *type* to `rpc_$register_mgr`.

If a server uses `rpc_$register_mgr` to register a manager for a specific interface and a specific type that is not nil, the server must use `rpc_$register_object` to register an object.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$op_rng_error	The requested operation does not correspond to a valid operation in the requested interface.
rpc_$unk_if	The requested interface is not known. It is not registered in the server, the version number of the registered interface is different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.
rpc_$too_many_ifs	The maximum number of interfaces is already registered with the RPC runtime library; the server must unregister some interface before it registers an additional interface.
rpc_$not_in_call	An internal error.
rpc_$you_crashed	This error can occur if a server has crashed and restarted. A client RPC runtime library sends the error to the server if the client makes a remote procedure call before the server crashes, then receives a response after the server restarts.
rpc_$proto_error	An internal protocol error.
rpc_$illegal_register	You are trying to register an interface that is already registered and you are using an EPV different from the one used when the interface was first registered. An interface can be multiply registered, but you must use the same EPV in each `rpc_$register` routine.

Files

`RPC$IDL:RPC.IDL`

See Also

rpc_$register, rpc_$register_object, rpc_$unregister

rpc_$register_object

Name

rpc_$register_object – register an object (server only)

Format

#include <rpc.h>

void rpc_$register_object(*object, type, status***)**
uuid_$t **object*;
uuid_$t **type*;
status_$t **status*;

Arguments

object	The UUID of the object being registered.
type	The UUID of the type of the object.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$register_object` routine declares that a server supports operations on a particular object and declares the type of that object.

A server must register objects with `rpc_$register_object` only if it registers generic interfaces with `rpc_$register_mgr`. When a server receives a call, the RPC runtime library searches for the object identified in the call (that is the object that the client specified in the handle) among the objects registered by the server. If the object is found, the type of the object determines which of the manager EPVs should be used to operate on the object.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$op_rng_error	The requested operation does not correspond to a valid operation in the requested interface.

rpc_$unk_if	The requested interface is not known. It is not registered in the server, the version number of the registered interface is different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.
rpc_$too_many_ifs	The maximum number of interfaces is already registered with the RPC runtime library; the server must unregister some interface before it registers an additional interface.
rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.
rpc_$illegal_register	You are trying to register an interface that is already registered and you are using an EPV different from the one used when the interface was first registered. An interface can be multiply registered, but you must use the same EPV in each `rpc_$register` routine.

Files

`RPC$IDL:RPC.IDL`

See Also

rpc_$register, rpc_$register_mgr, rpc_$unregister

rpc_$set_async_ack

Name

rpc_$set_async_ack – set or clear asynchronous-acknowledgement mode (client only)

Format

#include <rpc.h>

void rpc_$set_async_ack (*state*)
unsigned long *state*;

Arguments

state	If "true" (nonzero), asynchronous-acknowledgement mode is set. If "false" (zero), synchronous-acknowledgement mode is set.

Description

The `rpc_$set_async_ack` call sets or clears asynchronous-acknowledgement mode in a client.

Synchronous-acknowledgement mode is the default. Calling the routine with a nonzero value for *state* sets asynchronous-acknowledgement mode. Calling it with a zero value for *state* sets synchronous-acknowledgement mode.

After a client makes a remote procedure call and receives a reply from a server, the RPC runtime library at the client acknowledges its receipt of the reply. This "reply acknowledgement" can occur either synchronously (before the runtime library returns to the caller) or asynchronously (after the runtime library returns to the caller).

It is generally good to allow asynchronous reply acknowledgements. Asynchronous-acknowledgement mode can save the client runtime library from making explicit reply acknowledgements, because after a client receives a reply, it may shortly issue another call that can act as an implicit acknowledgement.

Asynchronous-acknowledgement mode requires that an "alarm" be set to go off sometime after the remote procedure call returns. Note that setting the alarm can cause two problems:

1 If only one alarm can be set, and the application itself may be trying to use it

2 If, at the time the alarm goes off, and the application is blocked in a system call that is doing I/O to a "slow device" (such as a terminal), the

system call will return an error (with the EINTR errno); the application may not be coded to expect this error.

If neither of these problems exists, set asynchronous-acknowledgement mode in the application to get greater efficiency.

Files

```
RPC$INCLUDE:RPC.H
RPC$IDL:RPC.IDL
```

rpc_$set_binding

Name

rpc_$set_binding – bind an RPC handle to a server (client only)

Format

#include <rpc.h>

void rpc_$set_binding(*handle, sockaddr, slength, status***)**
handle_t *handle***;**
socket_$addr_t **sockaddr***;**
unsigned long *slength***;**
status_$t **status***;**

Arguments

handle	An RPC handle.
sockaddr	The socket address of the server with which the handle is being associated.
slength	The length, in bytes, of *sockaddr*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$set_binding` routine sets the binding of an RPC handle to the specified server. The handle then identifies a specific object at a specific server. Any subsequent remote procedure calls that a client makes using the handle are sent to this destination.

You can use this routine either to set the binding in an unbound handle or to replace the existing binding in a fully bound or bound-to-host handle.

Example

The following statement sets the binding on the handle `h` to the first server in the `lbresults` array, which was returned by a previous Location Broker lookup routine, `lb_$lookup_interface`:

```
rpc_$set_binding (h, &lbresults[0].saddr, lbresults[0].saddr_len,
                             &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$cant_bind_sock	The RPC runtime library created a socket but was unable to bind it to a socket address.
rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

`RPC$IDL:RPC.IDL`

See Also

rpc_$alloc_handle, rpc_$clear_binding, rpc_$clear_server_binding

rpc_$set_fault_mode

Name

rpc_$set_fault_mode – set the fault-handling mode for a server (server only)

Format

#include <rpc.h>

unsigned long rpc_$set_fault_mode(*state***)**
unsigned long *state***;**

Arguments

state	If "true" (not zero), the server exits when a fault occurs. If "false" (zero), the server reflects faults back to the client.

Description

The `rpc_$set_fault_mode` function controls the handling of faults that occur in user server routines.

In the default mode, the server reflects faults back to the client and continues processing. Calling `rpc_$set_fault_mode` with value other than zero for *state* sets the fault-handling mode so that the server sends an **rpc_$comm_failure** fault back to the client and exits. Calling `rpc_$set_fault_mode` with *state* equal to zero resets the fault-handling mode to the default.

This function returns the previous state of the fault-handling mode.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine.

rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

`RPC$IDL:RPC.IDL`

Name

rpc_$set_short_timeout – set or clear short-timeout mode (client only)

Format

#include <rpc.h>

unsigned long rpc_$set_short_timeout(*handle, state, status***)**
handle_t *handle***;**
unsigned long *state***;**
status_$t **status***;**

Arguments

handle An RPC handle.

on If "true" (not zero), short-timeout mode is set on *handle*. If "false" (zero), standard timeouts are set.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$set_short_timeout` routine sets or clears short-timeout mode on a handle. If a client uses a handle in short-timeout mode to make a remote procedure call, but the server does not respond, the call fails quickly. As soon as the server responds, standard timeouts take effect and apply for the remainder of the call.

Calling `rpc_$set_short_timeout` with a value other than zero for *state* sets short-timeout mode. Calling it with *state* equal to zero, sets standard timeouts. Standard timeouts are the default.

This routine returns the previous setting of the timeout mode in `status.all`.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call An internal error.

rpc_$proto_error An internal protocol error.

Files

```
RPC$IDL:RPC.IDL
```

Name

rpc_$shutdown – shut down a server (server only)

Format

#include <rpc.h>

void rpc_$shutdown(*status***)**
status_$t ***status*;**

Arguments

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$shutdown` routine shuts down a server. When this routine is executed, the server stops processing incoming calls and `rpc_$listen` returns.

If `rpc_$shutdown` is called from within a remote procedure, that procedure completes, and the server shuts down after replying to the caller.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$comm_failure	The call could not be completed due to a communication problem.
rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

`RPC$IDL:RPC.IDL`

See Also

rpc_$allow_remote_shutdown, rpc_$listen, rrpc_$shutdown

rpc_$sockaddr_to_name

Name

rpc_$sockaddr_to_name – convert a socket address to a host name and port number (client or server)

Format

#include <rpc.h>

void rpc_$sockaddr_to_name(*&sockaddr, slength, name, &nlength, port, status*)
socket_$addr_t **sockaddr*;
unsigned long *slength*;
unsigned char *name*;
unsigned long **nlength*;
unsigned long **port*;
status_$t **status*;

Arguments

sockaddr	A socket address.
slength	The length, in bytes, of *sockaddr*.
name	A string that contains the host name and the address family. The format is *family:host [port]*, where *family* is **ip**.
nlength	On input, *nlength* is the length of the *name* buffer. On output, *nlength* is the number of characters returned in the *name* parameter.
port	The socket port number.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$sockaddr_to_name` routine provides the address family, the host name, and the port number identified by the specified socket address.

NOTE

This routine has been superseded by the `socket_$to_name` routine.

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

`RPC$IDL:RPC.IDL`

See Also

rpc_$name_to_sockaddr, socket_$to_name

rpc_$unregister

Name

rpc_$unregister – unregister an interface (server only)

Format

#include <rpc.h>

void rpc_$unregister(*ifspec, status***)**
rpc_$if_spec_t ****ifspec*;**
status_$t ****status*;**

Arguments

ifspec	An **rpc_$if_spec_t**. An interface specifier obtained from a previous RPC register call. The interface being unregistered.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$unregister` routine unregisters an interface that the server previously registered with the RPC runtime library. After an interface is unregistered, the RPC runtime library will not pass requests for that interface to the server.

If a server uses several `rpc_$register` or `rpc_$register_mgr` routines to register an interface more than once, then it must call `rpc_$unregister` an equal number of times to unregister the interface.

Example

The following statement unregisters a matrix arithmetic interface:

```
rpc_$unregister (&matrix_$if_spec, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$op_rng_error	The requested operation does not correspond to a valid operation in the requested interface.
rpc_$unk_if	The requested interface is not known. It is not registered in the server, the version number of the registered interface is

	different from the version number specified in the request, or the UUID in the request does not match the UUID of the registered interface.
rpc_$not_in_call	An internal error.
rpc_$proto_error	An internal protocol error.

Files

`RPC$IDL:RPC.IDL`

See Also

rpc_$register, rpc_$register_mgr, rpc_$register_object

rpc_$use_family

Name

rpc_$use_family – create a socket of a specified address family for a remote procedure call (RPC) server (server only)

Format

#include <rpc.h>

void rpc_$use_family(*family, sockaddr, slength, status***)**
unsigned long *family***;**
socket_$addr_t **sockaddr***;**
unsigned long **slength***;**
status_$t **status***;**

Arguments

family	The address family of the socket to be created. The value must be one of **socket_$internet** or **socket_$unspec**.
sockaddr	The socket address of the socket on which the server will listen.
slength	The length, in bytes, of *sockaddr*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$use_family` routine creates a socket for a server without specifying its port number. The RPC runtime software assigns a port number. If a server must listen on a particular well-known port, use `rpc_$use_family_wk` to create the socket.

A server listens on one socket per address family, regardless of how many interfaces that it exports. Therefore, servers should make this call once per supported address family.

Example

The following statement creates a server's socket:

```
rpc_$use_family (family, &saddr, &slen, &status);
```

Diagnostics

This section lists status codes for errors returned by this rpc_$ routine in status.all.

rpc_$cant_create_sock
The RPC runtime library was unable to create a socket.

rpc_$not_in_call
An internal error.

rpc_$proto_error
An internal protocol error.

rpc_$too_many_sockets
The server is trying to use more than the maximum number of sockets that is allowed; it has called rpc_$use_family or rpc_$use_family_wk too many times.

rpc_$addr_in_use
The address and port specified in an rpc_$use_family_wk routine are already in use. This is caused by multiple calls to rpc_$use_family_wk with the same well-known port.

Files

RPC$IDL:RPC.IDL

See Also

rpc_$use_family_wk

rpc_$use_family_wk

Name

rpc_$use_family_wk – create a socket with a well-known port for a remote procedure call (RPC) server (server only)

Format

#include <rpc.h>

void rpc_$use_family_wk(*family, ifspec, sockaddr, slength, status*)
unsigned long *family*;
rpc_$if_spec_t **ifspec*;
socket_$addr_t **sockaddr*;
unsigned long **slength*;
status_$t **status*;

Arguments

family — The address family of the socket to be created. This value corresponds to the communications protocol used to access the socket and determines how the sockaddr is expressed. The value must be one of **socket_$unspec** or **socket_$internet.**

ifspec — The interface that will be registered by the server. Typically, this parameter is the interface *if_spec* generated by the NIDL compiler from the interface definition; the well-known port is specified as an interface attribute.

sockaddr — The socket address of the socket on which the server will listen.

slength — The length, in bytes, of *sockaddr*.

status — The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rpc_$use_family_wk` routine creates a socket that uses the port specified through the *if_spec* parameter. Use this routine to create a socket only if a server must listen on a particular well-known port. Otherwise, use `rpc_$use_family.`

A server listens on one socket per address family, regardless of how many interfaces that it exports. Therefore, servers that use well-known ports should make this call once per supported address family.

Example

The following statement creates the well-known socket identified by `sockaddr` for an array processor server:

```
rpc_$use_family_wk (socket_$internet, &matrix$if_spec,
                    &sockaddr, &slen, &status);
```

Diagnostics

This section lists status codes for errors returned by this `rpc_$` routine in `status.all`.

rpc_$cant_create_sock
: The RPC runtime library was unable to create a socket.

rpc_$not_in_call
: An internal error.

rpc_$proto_error
: An internal protocol error.

rpc_$too_many_sockets
: The server is trying to use more than the maximum number of sockets that is allowed; it has called `rpc_$use_family` or `rpc_$use_family_wk` too many times.

rpc_$bad_pkt
: The server or client has received an ill-formed packet.

rpc_$addr_in_use
: The address and port specified in an `rpc_$use_family_wk` routine are already in use. This is caused by multiple calls to `rpc_$use_family_wk` with the same well-known port.

Files

`RPC$IDL:RPC.IDL`

See Also

rpc_$use_family

rrpc_$ Routine Reference Pages 13

This chapter contains reference pages for the `rrpc_$` routines, which enable a client to request information about a server or to shut down a server.

The `rrpc_$` interface is defined by these files:

On VMS systems	RPC$IDL:RRPC.IDL
On ULTRIX systems	`/usr/include/idl/rrpc.idl`

13.1 Constants

This section describes constants used in **rrpc_$** calls.

rrpc_$sv
Indices for elements in an **rrpc_$stat_vec_t** array. Each element is a 32-bit integer representing a statistic about a server. The following list describes the statistic indexed by each **rrpc_$sv** constant:

rrpc_$sv_calls_in
The number of calls processed by the server.

rrpc_$sv_rcvd
The number of packets received by the server.

rrpc_$sv_sent
The number of packets sent by the server.

rrpc_$sv_calls_out
The number of calls made by the server.

rrpc_$sv_frag_resends
The number of fragments sent by the server that duplicated previous sends.

rrpc_$sv_dup_frags_rcvd
The number of duplicate fragments received by the server.

status_$ok
A constant used to check status. If a completion status is equal to **status_$ok**, then the system call that supplied it was successful.

13.2 Data Types

This section describes data types used in `rpc_$` routines.

handle_t An RPC handle.

rrpc_$interface_vec_t
An array of **rpc_$if_spec_t**, RPC interface specifiers.

rrpc_$stat_vec_t
An array of 32-bit integers, indexed by `rrpc_$sv` constants, representing statistics about a server.

rpc_$if_spec_t
An RPC interface specifier. An opaque data type containing information about an interface, including the UUID, the version number, the number of operations in the interface, and any well known ports used by servers that export the interface.

Applications may need to access two members of **rpc_$if_spec_t**:

id A **uuid_$t** indicating the interface UUID.

vers An unsigned 32-bit integer indicating the interface version.

Name

rrpc_$are_you_there – check whether a server is answering requests

Format

#include <rrpc.h>

void rrpc_$are_you_there(*handle*, ******status***)**
handle_t *handle***;**
status_$t ******status***;**

Arguments

handle	A remote procedure call (RPC) handle.
status	The completion status.

Description

The `rrpc_$are_you_there` call checks whether a server is answering requests.

Restrictions

On the client side, because of the way the `rrpc_` calls are defined and implemented in the run-time library `libnck.a`, you must explicitly call into the entry point vector table for the `rrpc_` interface to send an `rrpc_` request across the network. The following is an example of a call that works as desired.

```
(*rrpc_$client_epv.rrpc_$are_you_there)(handle, &status);
```

The server side stub routine calls the entry point `rrpc_$are_you_there` on behalf of the client. The results of the call are then passed back to the client.

Files

`RPC$IDL:RRPC.IDL`

rrpc_$inq_interfaces

Name

rrpc_$inq_interfaces – obtain a list of the interfaces that a server exports

Format

#include <rrpc.h>

void rrpc_$inq_interfaces(*handle, max_ifs, ifs, l_if, status***)**
handle_t *handle***;**
unsigned long *max_ifs***;**
rrpc_$interface_vec_t *ifs[ ]***;**
unsigned long **l_if***;**
status_$t **status***;**

Arguments

handle	An RPC handle.
max_ifs	The maximum number of elements in the array of interface specifiers.
ifs	An array of **rpc_$if_spec_t**.
l_if	The index of the last element in the returned array.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rrpc_$inq_interfaces` routine returns an array of RPC interface specifiers.

Restrictions

On the client side, because of the way the `rrpc_` calls are defined and implemented in the run-time library `libnck.a`, you must explicitly call into the entry point vector table for the `rrpc_` interface to send an `rrpc_` request across the network. The following is an example of a call that works as desired:

```
(*rrpc_$client_epv.rrpc_$inq_interfaces)(handle,
            (unsigned long) max_ifs, ifs, &l_if, &status);
```

The server side stub routines call the entry point `rrpc_$inq_interfaces` on behalf of the client. The results of the call are then passed back to the client.

Files

RPC$IDL:RRPC.IDL

rrpc_$inq_stats

Name

rrpc_$inq_stats – obtain statistics about a server

Format

#include <rrpc.h>

void rrpc_$inq_stats(*handle, max_stats, stats, l_stat, status***)**
handle_t *handle***;**
unsigned long *max_stats***;**
rrpc_$stat_vec_t *stats***;**
unsigned long **l_stat***;**
status_$t **status***;**

Arguments

handle A remote procedure call (RPC) *handle* .

max_stats The maximum number of elements in the array of statistics.

stats An array of 32-bit integers representing statistics about the server. A set of **rrpc_$sv** constants defines indices for the elements in this array. The following list describes the statistic indexed by each **rrpc_$sv** constant:

rrpc_$sv_calls_in
The number of calls processed by the server.

rrpc_$sv_rcvd
The number of packets received by the server.

rrpc_$sv_sent
The number of packets sent by the server.

rrpc_$sv_calls_out
The number of calls made by the server.

rrpc_$sv_frag_resends
The number of fragments sent by the server that duplicated previous sends.

rrpc_$sv_dup_frags_rcvd
The number of duplicate fragments received by the server.

l_stat The index of the last element in the returned array.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rrpc_$inq_stats` routine returns an array of integer statistics about a server.

Restrictions

On the client side, because of the way the `rrpc_` calls are defined and implemented in the run-time library `libnck.a`, you must explicitly call into the entry point vector table for the `rrpc_` interface to send an `rrpc_` request across the network. The following is an example of a call that works as desired:

```
(*rrpc_$client_epv.rrpc_$inq_stats)(handle,
        (unsigned long) max_stats, stats, &l_stat ,&status);
```

The server sidestub routine calls the entry oint `rrpc_$inq_stats` on behalf of the client. The results of the call are then passed back to the client.

Files

`RPC$IDL:RRPC.IDL`

rrpc_$shutdown

Name

rrpc_$shutdown – shut down a server

Format

#include <rrpc.h>

void rrpc_$shutdown(*handle, status***)**
handle_t *handle***;**
status_$t **status***;**

Arguments

handle A remote procedure call (RPC) handle.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `rrpc_$shutdown` routine shuts down a server, if the server allows it. A server can use the `rpc_$allow_remote_shutdown` routine to allow or disallow remote shutdown.

Restrictions

On the client side, because of the way the `rrpc_` calls are defined and implemented in the run-time library `libnck.a`, you must explicitly call into the entry point vector table for the `rrpc_` interface to send an `rrpc_` request across the network. The following is an example of a call that works as desired:

```
(*rrpc_$client_epv.rrpc_$shutdown)(handle, &status);
```

The server side stub routine calls the entry point `rrpc_$shutdown` on behalf of the client. The results of the call are then passed back to the client.

Before making the shutdown call, the server must have previously executed the following call:

```
rpc_$allow_remote_shutdown((unsigned long) TRUE,NULL,&status);
```

If the server has not allowed remote shutdown, the `rrpc_$shutdown` call returns an `operation not implemented` status code.

Files

RPC$IDL:RRPC.IDL

See Also

rpc_$allow_remote_shutdown, rpc_$shutdown

socket_$ Routine Reference Pages 14

This chapter contains reference pages for the `socket_$` routines, which manipulate socket addresses. Unlike the routines that operating systems such as BSD UNIX provide, the `socket_$` routines operate on addresses of any protocol family.

The `socket_$` interface. is defined by these files:

On VMS systems	RPC$IDL:SOCKET.IDL
On ULTRIX systems	`/usr/include/idl/socket.idl`

14.1 Constants

This section describes constants used in `socket_$` routines.

socket_$eq Flags indicating the fields to be compared in a **socket_$equal** call.

socket_$eq_hostid
Indicates that the host IDs are to be compared.

socket_$eq_netaddr
Indicates that the network addresses are to be compared.

socket_$eq_port
Indicates that the port numbers are to be compared.

socket_$eq_network
Indicates that the network IDs are to be compared.

socket_$unspec_port
A port number indicating to the RPC run-time library that no port is specified.

socket_$addr_family_t
Values used to specify the address family in a **socket_$addr_t** structure. Note that several of the `rpc_$` and `socket_$` routines use the 32-bit-integer equivalents of these values.

socket_$unspec
Address family is unspecified.

socket_$internet
Internet Protocols (IP).

status_$ok A constant used to check status. If a completion status is equal to **status_$ok**, then the routine that supplied it was successful.

14.2 Data Types

This section describes data types used in `socket_$` routines.

socket_$addr_family_t
An enumerated type for specifying an address family. The Constants section lists values for this type.

socket_$addr_list_t
An array of socket addresses in **socket_$addr_t** format.

socket_$addr_t
A structure that uniquely identifies a socket address. This structure consists of a **socket_$addr_family_t** specifying an address family and 14 bytes specifying a socket address.

socket_$host_id_t
A structure that uniquely identifies a host. This structure consists of a **socket_$addr_family_t** specifying an address family and 12 bytes specifying a host.

socket_$len_list_t
An array of unsigned 32-bit integers, the lengths of socket addresses in a **socket_$addr_list_t**.

socket_$local_sockaddr_t
An array of 50 characters, used to store a socket address in a format native to the local host.

socket_$net_addr_t
A structure that uniquely identifies a network address. This structure consists of a **socket_$addr_family_t** specifying an address family and 12 bytes specifying a network address. It contains both a host ID and a network ID.

socket_$string_t
An array of 100 characters, used to store the string representation of an address family or a socket address.

The string representation of an address family is a textual name such as **dds**, **ip**, or **unspec**.

The string representation of a socket address has the format *family*:*host* [*port*], where *family* is the textual name of an address family, *host* is either a textual host name or a numeric host ID preceded by a #, and *port* is a port number.

status_$t

A status code. Most of the DECrpc routines supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the routines can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all All 32 bits in the status code. If **all** is equal to **status_$ok**, the routine that supplied the status was successful.

fail If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys This indicates the subsystem that encountered the error.

modc This indicates the module that encountered the error.

code This is a signed number that identifies the type of error that occurred.

socket_$equal

Name

socket_$equal – compare two socket addresses

Format

#include <socket.h>

boolean socket_$equal(*&sockaddr1, s1length, &sockaddr2, s2length, flags, status*)
socket_$addr_t **sockaddr1*;
unsigned long *s1length*;
socket_$addr_t **sockaddr2*;
unsigned long *s2length*;
unsigned long *flags*;
status_$t **status*;

Arguments

sockaddr1 — A socket address. The socket address is the structure returned by either `rpc_$use_family` or `rpc_$use_family_wk`.

s1length — The length, in bytes, of *sockaddr1*.

sockaddr2 — A socket address. The socket address is the structure returned by either `rpc_$use_family` or `rpc_$use_family_wk`.

s2length — The length, in bytes, of *sockaddr2*.

flags — The logical OR of values selected from the following:

socket_$eq_hostid	Indicates that the host IDs are to be compared.
socket_$eq_netaddr	Indicates that the network addresses are to be compared.
socket_$eq_port	Indicates that the port numbers are to be compared.
socket_$eq_network	Indicates that the network IDs are to be compared.

status — The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The socket_$equal routine compares two socket addresses. The *flags* parameter determines which fields of the socket addresses are compared. The call returns "true" (not zero) if all of the fields compared are equal, "false" (zero) if not.

Example

The following routine compares the network and host IDs in the socket addresses *sockaddr1* and *sockaddr2:*

```
if (socket_$equal (&sockaddr1, s1length, &sockaddr2, s2length,
        socket_$eq_network | socket_$eq_hostid, status))
printf ("sockaddrs have equal network and host IDs\n");
```

Files

RPC$IDL:SOCKET.IDL
RPC$INCLUDE:SOCKET.H

socket_$family_from_name

Name

socket_$family_from_name – convert an address family name to an integer

Format

#include <socket.h>

unsigned long socket_$family_from_name(*name, nlength, status***)**
socket_$string_t *name***;**
unsigned long *nlength***;**
status_$t **status***;**

Arguments

name	The textual name of an address family. Currently, only **ip** is supported.
nlength	The length, in bytes, of *name*.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `socket_$family_from_name` routine returns the integer representation of the address family specified in the text string *name*.

Example

The server program for the banks example, `/usr/examples/banks/bankd.c` accepts a textual family name as its first argument. The program uses the following `socket_$family_from_name` routine to convert this name to the corresponding integer representation:

```
family = socket_$family_from_name
             (argv[1], (long)strlen(argv[1]), &status);
```

Files

```
RPC$IDL:SOCKET.IDL
RPC$INCLUDE:SOCKET.H
```

See Also

socket_$family_to_name, socket_$from_name, socket_$to_name

socket_$family_to_name

Name

socket_$family_to_name – convert an integer address family to a textual name

Format

#include <socket.h>

void socket_$family_to_name(*family, name, nlength, status***)**
unsigned long *family***;**
socket_$string_t *name***;**
unsigned long **nlength***;**
status_$t **status***;**

Arguments

family	The integer representation of an address family.
name	The textual name of *family*. Currently, only **ip** is supported.
nlength	On input, the maximum length, in bytes, of the name to be returned. On output, the actual length of the returned name.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `socket_$family_to_name` routine converts the integer representation of an address family to a textual name for the family.

Files

```
RPC$IDL:SOCKET.IDL
RPC$INCLUDE:SOCKET.H
```

Name

socket_$from_name – convert a name and port number to a socket address

Format

#include <socket.h>

void socket_$from_name(*family, name, nlength, port, &sockaddr, &slength, status***)**
unsigned long *family***;**
socket_$string_t *name***;**
unsigned long *nlength***;**
unsigned long *port***;**
socket_$addr_t **sockaddr***;**
unsigned long **slength***;**
status_$t **status***;**

Arguments

family The integer representation of an address family. Value can be **socket_$internet** or **socket_$unspec**. If the *family* parameter is **socket_$unspec**, then the *name* parameter is scanned for a prefix of *family*: (for example, **ip:**).

name A string in the format *family*:*host* [*port*], where *family*:, *host*, and [*port*] are all optional.

The *family* is an address family. The only valid *family* is **ip**. If you specify a *family* as part of the *name* parameter, you must specify **socket_$unspec** in the *family* parameter.

The *host* is a host name. Use a leading number sign (#) to indicate that the host name is in the standard numeric form (for example, #192.9.8.7). If *host* is omitted, the local host name is used.

The *port* is a port number. If you specify a *port* as part of the *name* parameter, the *port* parameter is ignored.

nlength
The length, in bytes, of *name*.

port A port number. If you specify a port number in the *name* parameter, this parameter is ignored.

sockaddr
A socket address.

slength The length, in bytes, of *sockaddr*.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `socket_$from_name` routine converts a textual address family, host name, and port number to a socket address. The address family and the port number can be either specified as separate parameters or included in the *name* parameter.

Files

```
RPC$IDL:SOCKET.IDL
RPC$INCLUDE:SOCKET.H
```

See Also

socket_$family_from_name, socket_$to_name

Name

socket_$to_name – convert a socket address to a name and port number

Format

#include <socket.h>

void socket_$to_name(*sockaddr, slength, name, nlength, port, status*)
socket_$addr_t **sockaddr*;
unsigned long *slength*;
socket_$string_t *name*;
unsigned long **nlength*;
unsigned long **port*;
status_$t **status*;

Arguments

sockaddr	A socket address. The socket address is the structure returned by either `rpc_$use_family` or `rpc_$use_family_wk`.
slength	The length, in bytes, of *sockaddr*.
name	A string in the format *family:host[port]*, where *family* is the address family and *host* is the host name; *host* can be in the standard numeric form (for example, #192.1.2.3) if a textual host name cannot be obtained. Currently, only **ip** is supported for *family*.
nlength	On input, the maximum length, in bytes, of the name to be returned. On output, the actual length of the name returned.
port	The port number.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `socket_$to_name` routine converts a socket address to a textual address family, host name, and port number.

socket_$to_name

Files

```
RPC$IDL:SOCKET.IDL
RPC$INCLUDE:SOCKET.H
```

See Also

socket_$family_to_name, socket_$from_name, socket_$to_numeric_name

Name

socket_$to_numeric_name – convert a socket address to a numeric name and port number

Format

#include <socket.h>

void socket_$to_numeric_name(*&sockaddr, slength, name, &nlength, &port, status***)**

socket_$addr_t **sockaddr*;
unsigned long *slength*;
socket_$string_t *name*;
unsigned long **nlength*;
unsigned long **port*;
status_$t **status*;

Arguments

sockaddr	A socket address. The socket address is the structure returned by either `rpc_$use_family` or `rpc_$use_family_wk`.
slength	The length, in bytes, of *sockaddr*.
name	A string in the format *family:host[port]*, where *family* is the address family and *host* is the host name in the standard numeric form (for example, #192.7.8.9 for an IP address). Currently only **ip** is supported for *family*.
nlength	On input, the maximum length, in bytes, of the name to be returned (error if less than size of "nnnnn.nnnn"). On output, the actual length of the name returned.
port	The port number.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

socket_$to_numeric_name

Description

The `socket_$to_numeric_name` routine converts a socket address to a textual address family, a numeric host name, and a port number.

Files

```
RPC$IDL:SOCKET.IDL
RPC$INCLUDE:SOCKET.H
```

See Also

socket_$family_to_name, socket_$from_name, socket_$to_name

Name

socket_$valid_families – obtain a list of valid address families

Format

#include <socket.h>

void socket_$valid_families(*max_families, families, status*)
unsigned long **max_families*;
socket_$addr_family_t *families*[ ];
status_$t **status*;

Arguments

max_families	The maximum number of families that can be returned.
families[ ]	An array of **socket_$addr_family_t**. Possible values for this type are enumerated in `/idl/nbase.idl`. Currently, only **ip** is supported for *family*.
status	The completion status. This variable is set if the *families[ ]* array is not long enough to hold all the valid families. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `socket_$valid_families` routine returns a list of the address families that are valid on the calling host.

Example

The following call returns the valid address family:

```
socket_$valid_families (1, &families, $status);
```

Files

```
RPC$IDL:SOCKET.IDL
RPC$INCLUDE:SOCKET.H
```

socket_$valid_families

See Also

socket_$valid_family

Name

socket_$valid_family – check whether an address family is valid

Format

#include <socket.h>

boolean socket_$valid_family (*family, status*)
unsigned long *family*;
status_$t **status*;

Arguments

family	The integer representation of an address family.
status	The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `socket_$valid_family` routine returns "true" if the specified address family is valid for the calling host, "false" if not valid.

Example

The following routine checks whether **socket_$internet** is a valid address family:

```
internetvalid = socket_$valid_family(socket_$internet, &status);
```

Files

```
RPC$IDL:SOCKET.IDL
RPC$INCLUDE:SOCKET.H
```

See Also

socket_$valid_families

uuid_$ Routine Reference Pages 15

This chapter contains reference pages for the `uuid_$` routines, which operate on UUIDs (Universal Unique Identifiers).

The `uuid_$` interface is defined by these files:

On VMS systems	RPC$IDL:UUID.IDL
On ULTRIX systems	`/usr/include/idl/uuid.idl`

15.1 External Variables

This section describes external variables used in `uuid_$` routines.

uuid_$nil

An external **uuid_$t** variable that is preassigned the value of the nil UUID. Do not change the value of this variable.

15.2 Data Types

This section describes data types used in `uuid_$` routines.

status_$t

A status code. Most of the DECrpc routines supply their completion status in this format. The **status_$t** type is defined as a structure containing a long integer:

```
struct status_$t {
    long all;
    }
```

However, the routines can also use **status_$t** as a set of bit fields. To access the fields in a returned status code, you can assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

all — All 32 bits in the status code. If **all** is equal to **status_$ok**, the routine that supplied the status was successful.

fail — If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

subsys — This indicates the subsystem that encountered the error.

modc — This indicates the module that encountered the error.

code — This is a signed number that identifies the type of error that occurred.

uuid_$string_t

A string of 37 characters (including a null terminator) that is an ASCII representation of a UUID. The format is *cccccccccccc*.*ff*.*h1*.*h2*.*h3*.*h4*.*h5*.*h6*.*h7*, where *cccccccccccc* is the timestamp, *ff* is the address family, and *h1* ... *h7* are the 7 bytes of host identifier. Each character in these fields is a hexadecimal digit.

uuid_$t A 128-bit value that uniquely identifies an object, type, or interface for all time. The **uuid_$t** type is defined as follows:

```
typedef struct uuid_$t {
    unsigned long time_high;
    unsigned short time_low;
    unsigned short reserved;
    unsigned char family;
    unsigned char (host)[7];
} uuid_$t;
```

time_high — The high 32 bits of a 48-bit unsigned time value that is the number of 4-microsecond intervals that have passed between 1 January 1980 00:00 GMT and the time of UUID creation.

time_low — The low 16 bits of the 48-bit time value.

reserved — 16 bits of reserved space.

family — 8 bits identifying an address family.

host — 7 bytes identifying the host on which the UUID was created. The format of this field depends on the address family.

15.3 Example

The following routine returns as `foo_uuid` the UUID corresponding to the character-string representation in `foo_uuid_rep`:

```
uuid_$decode (foo_uuid_rep, &foo_uuid, &status);
```

uuid_$decode

Name

uuid_$decode – convert a character-string representation of a UUID into a UUID structure

Format

#include <uuid.h>

void uuid_$decode(*s, uuid, status*)
uuid_$string_t *s*;
uuid_$t **uuid*;
status_$t **status*;

Arguments

s The character-string representation of a UUID.

uuid The UUID that corresponds to *s*.

status The completion status. If the completion status returned in `status.all` is equal to **status_$ok**, then the routine that supplied it was successful.

Description

The `uuid_$decode` routine returns the UUID corresponding to a valid character-string representation of a UUID.

Example

The following routine returns as **foo_uuid** the UUID corresponding to the character-string representation in **foo_uuid_$rep:**

```
uuid_$decode (foo_uuid_$rep, &foo_uuid, &status);
```

Files

```
RPC$IDL:UUID.IDL
RPC$INCLUDE:UUID.H
```

See Also

uuid_$encode

Name

uuid_$encode – convert a UUID into its character-string representation

Format

#include <uuid.h>

void uuid_$encode(*uuid, s*)
uuid_$t **uuid*;
uuid_$string_t *s*;

Arguments

uuid	A UUID.
s	The character-string representation of *uuid*.

Description

The `uuid_$encode` routine returns the character-string representation of a UUID.

Example

The following routine returns as **foo_uuid_$rep** the character-string representation for the UUID **foo_uuid:**

```
uuid_$encode (&foo_uuid, foo_uuid_$rep);
```

Files

```
RPC$IDL:UUID.IDL
RPC$INCLUDE:UUID.H
```

See Also

uuid_$decode

uuid_$equal

Name

uuid_$equal – compare two UUIDs

Format

#include <uuid.h>

boolean uuid_$equal(*u1, u2*)
uuid_$t **u1*;
uuid_$t **u2*;

Arguments

u1 A UUID.

u2 Another UUID.

Description

The `uuid_$encode` routine compares the UUIDs *u1* and *u2*. It returns "true" if they are equal, "false" if they are not.

Example

The following code compares the UUIDs **bar_uuid** and **foo_uuid:**

```
if (uuid_$equal (bar_uuid, foo_uuid))
    printf ("bar and foo UUIDs are equal\n");
else
    printf ("bar and foo UUIDs are not equal\n");
```

Files

```
RPC$IDL:UUID.IDL
RPC$INCLUDE:UUID.H
```

Name

uuid_$gen – generate a new UUID

Format

#include <uuid.h>

void uuid_$gen (*uuid*)
uuid_$t **uuid*;

Arguments

uuid A pointer to a UUID structure to be filled in.

Description

The `uuid_$gen` routine returns a new UUID. Typically used when creating a new remote application.

Example

The following routine returns as **new_uuid** a new UUID:

```
uuid_$gen (&new_uuid);
```

Files

```
RPC$IDL:UUID.IDL
RPC$INCLUDE:UUID.H
```

Process and Utility Reference Pages 16

This chapter contains reference pages for the DECrpc processes and utilities.

lb_$admin

Name

lb_$admin – Location Broker Administrative Tool

Format

lb_$admin [**–version**] [**–nq**]

Note

This tool is available on VMS systems, ULTRIX systems, and other versions of the UNIX operating system. The command interface is common across all these systems, and therefore is not in a traditional DCL style.

For this command, precede qualifiers with a hyphen (-), rather than the customary slash (/).

You must define each DECrpc command as a foreign command.

Description

The `lb_$admin` tool monitors and administers the registrations of DECrpc-based servers in Global Local Broker (GLB) or Local Location Broker (LLB) databases. A server registers Universal Unique Identifiers (UUIDs) specifying an object, a type, and an interface, along with a socket address specifying its location. A client can locate servers by issuing lookup requests to GLBs and LLBs.

In accepting input or displaying output, `lb_$admin` uses either character strings or descriptive textual names to identify objects, types, and interfaces. A character string directly represents the data in a UUID in the following format, where each *n* is a hexadecimal digit:

nnnnnnnnnnnn.nn.nn.nn.nn.nn.nn.nn.nn

With `lb_$admin`, you examine or modify only one database at a time, referred to as the current database. The `use_broker` command selects the type of Location Broker database, GLB or LLB. The `set_broker` command selects the host whose LLB database is to be accessed.

Information about the `lb_$admin` commands is available through the `help` command.

Before invoking the `lb_admin` utility, define it as a foreign command in SYS$MANAGER:SYSLOGIN.COM, as shown in this example:

```
$ lb_admin :== rpc$exe:rpc$lb_admin.exe
```

Qualifiers

–nq Do not query for verification of wildcard expansions in `unregister` operations.

–version Display the version of the Network Computing Kernel (NCK) that this `lb_$admin` belongs to, but do not start the tool. (NCK is part of the Network Computing System (NCS) on which DECrpc is based.)

Commands

In the descriptions of `lookup`, `register`, and `unregister`, the *object*, *type*, and *interface* arguments can be either character strings representing UUIDs or textual names corresponding to UUIDs, as described earlier.

In the descriptions of `register` and `unregister`, the *location* argument is a string in the format *family:host[port]*, where *family* is an address family, *host* is a host name, and *port* is a port number. The only value for *family* is **ip**. You can use a leading number sign (#) to indicate that a host name is in the standard numeric form. For example, **ip:vienna[1756]**, and **ip:#192.5.5.5[1791]** are both acceptable *location* specifiers.

`a[dd]` Synonym for `register`.

`c[lean]` Find and delete obsolete entries in the current database.

When you issue the `clean` command, `lb_$admin` attempts to contact each server registered in the database. If the server does not respond, `lb_$admin` tries to look up its registration in the LLB database at the host where the server is located, tells you the result of this lookup, and asks whether you want to delete the entry. If a server responds, but its UUIDs do not match the entry in the database, `lb_$admin` tells you this result and asks whether you want to delete the entry, even if you used the **-nq** qualifier to `lb_$admin`.

There are two situations in which it is likely that a database entry should be deleted:

- The server does not respond, `lb_$admin` succeeds in contacting the LLB at the host where the server is located, and the server is not registered with that LLB. The server is probably no longer running.

- A server responds, but its UUIDs do not match the entry in the database. The server that responded is not the one that registered the entry.

Entries that meet either of these conditions are probably safe to delete and are considered eligible for automatic deletion (described in the next paragraph). In other situations, it is best not to delete the entry unless you can verify directly that the server is not running (for example, by listing the processes running on its host).

When the `clean` command asks whether you want to delete an entry, choose one of the following responses:

y[es] Delete the entry.

n[o] Leave the entry intact in the current database.

g[o] Invoke automatic deletion, in which all eligible entries (see the previous paragraph) are deleted and all ineligible entries are left intact, without your being queried, until all entries have been checked.

q[uit] Terminate the `clean` operation.

`d[elete]` Synonym for `unregister.`

`h[elp]` `[`*command*`]` or `?` `[`*command*`]`

Display a description of the specified *command* or, if none is specified, list all of the `lb_$admin` commands.

`l[ookup]` *object type interface*

Look up and display all entries with matching *object, type,* and *interface* fields in the current database. You can use asterisks as wildcards for any of the arguments. If all the arguments are wildcards, or if no arguments are given, `lookup` displays the entire database.

`q[uit]` Exit the `lb_$admin` session.

`r[egister]` *object type interface location annotation [flag]*

Add the specified entry to the current database. You can use an asterisk to represent the nil UUID in the *object, type*, and *interface* fields.

The *annotation* is a string of up to 64 characters annotating the entry. Use double quotation marks (" ") to delimit a string that contains a space or contains no characters. To embed a double quotation mark in the string, precede it with a backslash (\).

The *flag* is either `local` (the default) or `global`, indicating whether to mark the entry for local registration only or for registration in both the LLB and the GLB databases. The *flag* is a field that is stored with the entry; it does not affect where the entry is registered. The `set_broker` and `use_broker` commands select the particular LLB or GLB database for registration.

`set_broker` *[broker_switch] location*

Set the host for the current LLB or GLB. If you specify global as the *broker_switch*, `set_broker` sets the current GLB; otherwise, it sets the current LLB. The *host* is a *location* specifier as described earlier, but the *[port]* portion is ignored and can be omitted.

Issue the `use_broker` command, not the `set_broker` command, to determine whether subsequent operations will access the LLB or the GLB.

`set_t[imeout]` *[short / long]*

Set the timeout period used by `lb_$admin` for all of its operations. With an argument of `short` or `long`, `set_timeout` sets the timeout accordingly. With no argument, it displays the current timeout value.

`u[nregister]` *object type interface location*

Delete the specified entry from the current database.

You can use an asterisk as a wildcard in the *object, type,* and *interface* fields to match any value for the field. Unless you suppress queries by specifying the **-nq** qualifier of `lb_$admin`, `unregister` asks you whether to delete each matching entry. Choose one of the following responses:

y[es] Delete the entry.

n[o] Leave the entry in the database.

g[o] Delete all remaining database entries that match, without your being queried.

q[uit] Terminate the `unregister` operation, without deleting any more entries.

`us[e_broker]` *[broker_switch]*

Select the type of database that subsequent operations will access, GLB or LLB. The *broker_switch* is either `global` or `local`. If you do not supply a *broker_switch*, `use_broker` tells whether the current database is global or local.

Use `set_broker` to select the host whose GLB or LLB is to be accessed.

See Also

llbd, nrglbd
Guide to the Location Broker

Name

llbd – Local Location Broker Process

Format

llbd [**–version**]

Note

This tool is available on VMS systems, ULTRIX systems, and other versions of the UNIX operating system. The command interface is common across all these systems, and therefore is not in a traditional DCL style.

For this command, precede qualifiers with a hyphen (-), rather than the customary slash (/).

You must define each DECrpc command as a foreign command.

Description

The Local Location Broker Process (`llbd`) is part of the Network Computing System (NCS). It manages the Local Location Broker (LLB) database, which stores information about RPC-based server programs running on the local host.

A host must run `llbd` if it is to support the Location Broker forwarding function or to allow remote access (for example, by the `lb_$admin` tool) to the LLB database. In general, any host that runs an RPC-based server program must run an `llbd`, and `llbd` must be running before any such servers are started. Additionally, any network supporting RPC activity should have at least one host running a Global Location Broker Process (`nrglbd`).

The command file SYS$STARTUP:RPC$UCX_STARTUP.COM starts the `llbd` process on a VMS system. The process should run as a detached process, independently of login activity, for as long as the system is up.

The following example shows the line in the command procedure that starts the process:

```
$ RUN/DETACHED/PRIV=SYSPRV/PROCESS_NAME=RPC$LLBD RPC$EXE:RPC$LLBD.EXE
```

llbd

Qualifier

–version Display the version of the Network Computing Kernel (NCK) that this `llbd` belongs to, but do not start the process. (NCK is part of the Network Computing System (NCS) on which DECrpc is based.)

See Also

lb_$admin, nrglbd
Guide to the Location Broker

Name

nidl – Network Interface Definition Language Compiler

Format

nidl *filename* [*options*]

Note

This tool is available on VMS systems, ULTRIX systems, and other versions of the UNIX operating system. The command interface is common across all these systems, and therefore is not in a traditional DCL style.

For this command, precede qualifiers with a hyphen (-), rather than the customary slash (/).

You must define each DECrpc command as a foreign command.

Description

The `nidl` compiler is a compiler for the Network Interface Definition Language (NIDL).

The *filename* argument is the pathname of an interface definition file, written in the C syntax of NIDL.

The compiler generates a header file, a client stub file, a server stub file, and a client switch file, all in C source code. The compiler derives the names of these output files from *filename* by replacing the suffix (the rightmost period and all subsequent characters) with extensions for the client stub, server stub, and client switch.

Before invoking the `nidl` compiler, define it as a foreign command in SYS$MANAGER:SYSLOGIN.COM, as shown in this example:

```
$ nidl :== rpc$exe:rpc$nidl.exe
```

Qualifiers

–confirm Display the qualifiers chosen but do not compile anything. In displaying information about **–idir**, the compiler constructs the list of all directories it would use to resolve relative pathnames of imported files, not just the ones explicitly supplied. (If the list is empty, the compiler uses only the current directory.) This qualifier is useful for viewing the "idir list" and for viewing the default values for other qualifiers.

–cpp *pathname*
: Run the specified program instead of the default C preprocessor. You can use the **–confirm** qualifier to view the default pathname.

–def *def1* [*def2* ...]
: Pass the specified definitions to the C preprocessor. A definition can take either of two forms: *symbol* or *symbol=value*.

–exts *cstub–ext***,** *sstub–ext***,** *cswtch–ext*
: Set the extensions that the compiler uses to name the stub and switch files it generates. The text strings *cstub–ext*, *sstub–ext*, and *cswtch–ext* must be separated by commas, with no spaces; they are used as extensions for the client stub, the server stub, and the client switch, respectively. You can use the **–confirm** qualifier to view the defaults.

–f77c
: Generate client switch code that is compatible with the ULTRIX `f77` compiler. The NIDL Compiler appends an underscore (_) character to the name of each client switch routine, so that the routines can be called from FORTRAN programs generated by the `f77` compiler.

–f77s
: Generate server stub code that is compatible with the ULTRIX `f77` compiler. The NIDL Compiler appends an underscore (_) character to the name of each manager routine that the stub calls, so that the stub can call routines generated by the `f77` compiler.

–idir *directory1* [*directory2* ...]
: Use the specified directories as paths from which to resolve relative pathnames of imported files. The compiler generates an ordered list of these directories. By default, it prepends to this list your current working directory and appends the system `idl` directory. You can suppress this default by supplying the **–no_def_idir** qualifier.

–m
: Support multiple versions and multiple managers within a single server. This qualifier allows a server to export more than one version of an interface ("multiple versions") and to implement an interface for more than one type ("multiple managers").

The compiler appends the version number to the interface name when it generates identifiers in the stub and header files. For example, the interface specifier for version 3 of the **foobar** interface would be **foobar_v3$if_spec**.

The server for an interface compiled with **–m** must use `rpc_$register_mgr` to register its managers. The server supplies the name of a manager EPV to `rpc_$register_mgr;` the manager code defines this EPV. If the server supports objects of several types, it must use `rpc_$register_object` to register each object. These registrations enable the RPC runtime library at the server host to dispatch incoming requests to the correct manager.

If you do not specify either **–m** or its counterpart, **–s**, the compiler assumes **–s** and issues a warning. However, this default may be removed or changed in future NIDL compilers. Even if your server exports only one version of its interface and contains only one manager, use the **–m** qualifier, so that it will be easy for you to incorporate multiple versions and multiple managers later.

–no_cpp
Do not run the C preprocessor on the input file. If you specify this qualifier, the NIDL compiler does not interpret any C preprocessor statements (such as `#include` statements) in the interface definition.

–no_def_idir
Do not prepend the current working directory or append the system `idl` directory to the list of directories constructed from *-idir* arguments. If you specify **–no_def_idir** without **–idir**, the compiler resolves pathnames of imported files only relative to the current working directory.

–no_stubs
Do not generate any stub or switch files. The NIDL Compiler generates only header files and insert files.

–no_warn
Suppress warning messages.

–out *directory*
Place the generated files in *directory*. The default is the current working directory.

–s
Allow a server to export only a single version of an interface and to implement an interface for only a single type. This qualifier requests the behavior of NIDL compilers before

Version 1.5, which added support for multiple versions and multiple interfaces. (See the **–m** qualifier.)

The server for an interface compiled with **–s** must use `rpc_$register` to register its interfaces.

If you do not specify either **–s** or its counterpart, **–m**, the compiler assumes **–s** and issues a warning. However, this default may be removed or changed in future NIDL compilers. Even if your server exports only one version of its interface and contains only one manager, use the **–m** qualifier, so that it will be easy for you to incorporate multiple versions and multiple managers later.

–space_opt Reduce the size of generated stub code, possibly at the expense of slower data marshalling.

–version Display the version number of the NIDL compiler but do not compile anything or generate any output files.

See Also

uuid_gen

Name

nrglbd – Non-Replicating Global Location Broker Process

Format

nrglbd [**–version**]

Note

This tool is available on VMS systems, ULTRIX systems, and other versions of the UNIX operating system. The command interface is common across all these systems, and therefore is not in a traditional DCL style.

For this command, precede qualifiers with a hyphen (-), rather than the customary slash (/).

You must define each DECrpc command as a foreign command.

Description

The Global Location Broker (GLB), enables clients to locate servers on a network or internet. The GLB database stores the locations (that is, the network addresses and port numbers) where server processes are running. The GLB maintains this database and provides access to it.

The `nrglbd` daemon should run as a detached process. It requires no qualifiers or arguments. A Local Location Broker process (`llbd`) must be running on the local host when `nrglbd` is started.

You can run only one `nrglbd` on a network or internet.

The following command procedure starts the GLB process `nrglbd` on a VMS system:

```
SYS$STARTUP:RPC$UCX_STARTUP.COM
```

The process has no qualifiers and takes no arguments. It should run as a detached process, independently of login activity for as long as the system is up.

The following example shows the line in the command procedure that starts the process.

```
$ RUN/DETACHED/PRIV=SYSPRV/PROCESS_NAME=RPC$NRGLBD RPC$EXE:RPC$NRGLBD.EXE
```

nrglbd

Qualifier

–version Display the version of the Network Computing Kernel (NCK) that this `nrglbd` belongs to but do not start the process. (NCK is part of the Network Computing System (NCS) on which DECrpc is based.)

Restrictions

This section discusses the procedure to follow if the system running the `nrglbd` is taken off-line.

If you restart `nrglbd` on the same system and no server on any other system changed state, all things should run as before. If, however, an application tries to contact a server that is no longer running or which has different port numbers, the application will fail. The application also will not see any new server registrations.

If a copy of `glbdbase.dat` is not available, you must create an up to date version of the file before restarting `nrglbd`. To do so, use `lb_$admin` to query the `llbd` for registration data on every system running a DECrpc server and to register all DECrpc servers with the GLB on the new host. Then restart `nrglbd`.

See Also

lb_$admin, llbd
Guide to the Location Broker

Name

stcode – translate a hexadecimal status code value to a textual message

Format

stcode *hex_stat_code*

Note

This tool is available on VMS systems, ULTRIX systems, and other versions of the UNIX operating system. The command interface is common across all these systems, and therefore is not in a traditional DCL style.

For this command, precede qualifiers with a hyphen (-), rather than the customary slash (/).

You must define each DECrpc command as a foreign command.

Description

The `stcode` command prints the textual message associated with a hexadecimal status code. This command is useful when a program produces a hexadecimal status code instead of a textual message.

The `stcode` command processes predefined status codes. No provision is currently made to add user-defined status codes to the error text database.

Before running the `stcode` utility, define it as a foreign command in SYS$MANAGER:SYSLOGIN.COM, as shown in this example:

```
$ stcode :== rpc$exe:rpc$stcode.exe
```

Example

Translate the hexadecimal status code 1c010003:

```
# stcode 1c010003
unknown interface (network computing system/RPC runtime)
```

Files

RPC$EXE:RPC$STCODE.DAT

uuid_gen

Name

uuid_gen – UUID generating program

Format

uuid_gen [**–c**] [**–C**] [**–version**]

Note

This tool is available on VMS systems, ULTRIX systems, and other versions of the UNIX operating system. The command interface is common across all these systems, and therefore is not in a traditional DCL style.

For this command, precede qualifiers with a hyphen (-), rather than the customary slash (/).

You must define each DECrpc command as a foreign command.

Description

The `uuid_gen` program generates Universal Unique Identifiers (UUIDs). Without qualifiers, it generates a character-string representation of a UUID. The **–c** qualifier enables you to generate a template for Network Interface Definition Language (NIDL) files. The **–C** qualifier enables you to generate source-code representations of UUIDs, suitable for initializing variables of type `uuid_t`.

Before invoking the `uuid_gen` utility, define it as a foreign command in SYS$MANAGER:SYSLOGIN.COM, as shown in this example:

```
$ uuid_gen :== rpc$exe:rpc$uuid_gen.exe
```

Qualifiers

–c Generate a template, including a UUID attribute, for an interface definition in the C syntax of NIDL.

–C Generate a C source-code representation of a UUID.

–version Display the version of the Network Computing Kernel (NCK) that this `uuid_gen` belongs to, but do not generate a UUID. (NCK is part of the Network Computing System (NCS) on which DECrpc is based.)

Examples

Generate a character-string representation of a UUID:

```
$ uuid_gen
34dc23469000.0d.00.00.7c.5f.00.00.00
```

Generate a template for an interface definition in the C syntax of NIDL:

```
$ uuid_gen -c
%c
[
uuid(34dc239ec000.0d.00.00.7c.5f.00.00.00),
version(1)
]
interface INTERFACENAME {

}
```

Generate an interface definition template in the file `myfile.idl`:

```
$ define/user sys$output myfile.idl
$ uuid_gen -c
```

Generate a C source-code representation of a UUID:

```
$ uuid_gen -C
= { 0x34dc23af,
0xf000,
0x0000,
0x0d,
{0x00, 0x00, 0x7c, 0x5f, 0x00, 0x00, 0x00} };
```

Glossary

address family
: A set of communications protocols that use a common addressing mechanism to identify endpoints. The terms *address family* and *protocol family* are used synonymously in this manual.

allocate a handle
: To create a Remote Procedure Call (RPC) handle that identifies an object but not a location. Such a handle is said to be *allocated* or *unbound.*

attributes
: Characteristic declared in the Network Interface Definition Language (NIDL). An interface itself can be described by five attributes: uuid, local, version, port, implicit_handle. Type declarations and operation declarations also have specified attributes: type and field.

automatic binding
: Binding technique, in which the client uses generic handles that are then converted to Remote Procedure Call (RPC) handles by automatic binding routines. In an application that uses automatic binding, the client does not manage the binding. The handle variable is generic, and the application developer must supply autobinding and autounbinding routines that convert generic handles (used by the client) to RPC handles (used by the RPC run-time library). *See also* **binding state.**

binding
: The representation of a server in a handle. To bind a handle or to set its binding is to establish this representation. *See also* **binding state** and **handle.**

binding state
: The amount of information in a handle. A Remote Procedure Call (RPC) handle can exist in three binding states: unbound, bound-to-host, and fully bound.

binding technique
: Determines whether the client uses Remote Procedure Call (RPC) handles directly or uses generic handles that are then converted to RPC handles. *See also* **manual binding** and **automatic binding.**

bound-to-host handle
: Handle that identifies an object and a host but does not identify the port number of the server that exports the requested interface. When a client uses a bound-to-host handle to make a remote procedure call, the Remote Procedure Call (RPC) run-time library sends a message to the Local Location Broker (LLB) forwarding port on the specified host. The LLB forwards the message to the server.

bound-to-server handle
: *See* **fully bound** and **binding state**.

broadcast
: To send a remote procedure call to all hosts in a network.

broker
: A server that manages information resources, as in a Location Broker.

client
: A process that uses resources. In the context of this manual, a program that makes remote procedure calls.

entry point vector (EPV)
: A record of pointers to the operations in an interface.

explicit handle
: A handle that is passed as an operation parameter, rather than represented as a global variable in the client process. *See also* **implicit handle**.

export an interface
: To provide the operations defined by an interface. A server exports an interface to a client.

forward
: Automatic dispatch of a request to a server that exports the requested interface for the requested object. The Local Location Broker (LLB) forwards remote procedure calls that are sent to the LLB forwarding port on a server host.

fully bound handle
: A Remote Procedure Call handle that identifies an object, a host, and a port.

generic handle
: Handle variables that are not of type `handle_t`, such as a pathname. *See also* **RPC handle**.

GLB *See* **Global Location Broker**.

Global Location Broker (GLB)
: A server that maintains global information about objects on a network or an internet. Part of the Location Broker, it runs as the `nrglbd` process.

handle
A temporary local identifier for an object. A handle represents for a client process the object and a server that exports one or more interfaces to the object. A handle always represents the same object, but it may represent different servers at different times, or it may not specify a server at all. *See also* **binding**.

host
A computer that is attached to a network.

host ID
An identifier for a host. A host ID uniquely specifies a host within an address family on a network, but does not specify the network. A host ID may not be sufficient to establish communications with a host. *See also* **network ID**.

idempotent operation
An operation whose results do not affect the results of any operation. For example, a call that reads a value is idempotent, but an operation that increments a value is not.

implement an interface
To provide the routines that execute the operations in an interface. A manager implements one interface for one type.

implicit handle
A handle that is represented as a global variable in the client process, rather than passed as an operation parameter. *See also* **explicit handle**.

import an interface
To request the operations defined by an interface. A client imports an interface from a server. *See also* **export**.

interface
A set of operations defined by the Network Interface Definition Language (NIDL).

interface UUID
A Universal Unique Identifier (UUID) that permanently identifies a particular interface. Both the Remote Procedure Call (RPC) run-time library and the location broker use interface UUIDs to specify interfaces.

internet
A collection of networks interconnected by gateways.

LB *See* **Location Broker**.

LLB *See* **Local Location Broker**.

Local Location Broker (LLB)
A server that maintains information about objects on the local host. The LLB also provides the Location Broker forwarding facility.

Location Broker (LB)
: A set of software that includes the Local Location Broker, the Global Location Broker, and the Location Broker Client Agent. The Location Broker maintains information about the locations of objects.

Location Broker Client Agent
: Part of the Location Broker. Programs communicate with Global Location Brokers and Local Location Brokers by means of the Location Broker Client Agent.

manager
: A set of routines that implement the operations in one interface for objects of one type.

manual binding
: A binding technique in which the client uses Remote Procedure Call (RPC) handles.

marshall
: To copy data into a Remote Procedure Call (RPC) packet. Stubs perform marshalling. *See also* **unmarshall**.

network address
: A unique identifier (within an address family) for a specific host on a network or an internet. A network address is sufficient to identify a host, but it does not identify a communications endpoint within the host.

Network Computing System (NCS)
: A set of software components on which DECrpc is based. These components include the Remote Procedure Call run-time library, the Location Broker, and the NIDL Compiler.

Network Interface Definition Language (NIDL)
: A declarative language for the definition of interfaces. NIDL has two syntaxes, one resembling C and one resembling Pascal.

NIDL
: *See* **Network Interface Definition Language**.

NIDL Compiler
: An NCS tool that converts an interface definition written in Network Interface Definition Language (NIDL) into several program modules, including source code for client and server stubs. The NIDL Compiler accepts interface definitions written in either syntax of NIDL; it generates C source code and C or Pascal header files.

object
: An entity that is manipulated by well-defined operations. Disk files, printers, and array processors are examples of objects. Objects are accessed through interfaces. Every object has a type.

object UUID
: A Universal Unique Identifier (UUID) that identifies a particular object. Both the Remote Procedure Call (RPC) run-time library and the Location Broker use object UUIDs to identify objects.

opaque port
: A port that is dynamically assigned to a server by the Remote Procedure Call run-time library. The port number is said to be opaque because there is no need for either clients or servers to know the number. *See also* **well known port.**

operation
: A procedure through which an object is accessed.

port
: A specific communications endpoint within a host. A port is identified by a port number. *See also* **socket.**

port number
: One of the three parts in a socket address. For example, the character string *77* might represent a port number, while *ip:wooster[77]* might represent a socket address.

protocol family
: A set of communications protocols, for example, the DARPA Internetwork Protocols. All members of a protocol family use a common addressing mechanism to identify endpoints. The terms *address family* and *protocol family* are used synonymously in this manual.

register an interface
: To make an interface known to the Remote Procedure Call (RPC) run-time library and thereby available to clients through the RPC mechanism. The `rpc_$register` call registers an interface.

register a manager
: To make a manager (the code that implements a particular interface for a particular type) known to the Remote Procedure Call (RPC) run-time library and thereby available to clients through the RPC mechanism. The `rpc_$register_mgr` call registers a manager.

register an object with the Location Broker
: To enter an object and its location in the Location Broker database. The `lb_$register` call registers an object with the Location Broker. A program can use Location Broker lookup calls to determine the location of a registered object.

register with the RPC run-time library

Call to `rpc_$register` that allows your program to call routines in the Remote Procedure Call (RPC) run-time library. Initializes access to the run-time library.

remote procedure call

An invocation of a remote operation. You can make remote procedure calls between processes on different hosts or on the same host.

Remote Procedure Call (RPC) run-time library

The set of `rpc_$` system calls that DECrpc provides to implement its remote procedure call mechanism.

RPC *See* **Remote Procedure Call.**

RPC handle

A Remote Procedure Call (RPC) handle is a pointer to an opaque data structure containing the information needed to access an object. The name for this pointer type is `handle_t`.

server

A process that implements interfaces. In the context of this manual, a server whose procedures can be invoked from remote hosts. A server exports one or more interfaces for one or more objects.

set a binding

To set the representation of a server location in a Remote Procedure Call (RPC) handle.

signature

The syntax of an operation, that is, its name, the data type it returns, and the order and types of its parameters. The definition of an operation specifies only its signature, not its implementation.

socket

A communications endpoint in the form of a message queue. A socket is identified by a socket address.

socket address

A data structure that uniquely identifies a specific communications endpoint. A socket address consists of a port number and a network address.

stub

A program module that transfers remote procedure calls and responses between a client and a server. Stubs perform marshalling, unmarshalling, and data format conversion. Both clients and servers have stubs. The NIDL Compiler generates client and server stub code from an interface definition.

transmitted type

For data types with the `transmit_as` attribute, the data type that stubs pass over the network. Stubs invoke conversion routines to convert the transmitted type to a presented type, which is manipulated by clients and servers.

type

A class of object. All objects of a specific type can be accessed through the same interface or interfaces.

type UUID

A Universal Unique Identifier (UUID) that permanently identifies a particular type. Both the Remote Procedure Call (RPC) run-time library and the Location Broker use type UUIDs to specify types.

unbound handle

A Remote Procedure Call (RPC) handle that identifies an object but not a location. Synonymous with *allocated handle.*

Universal Unique Identifier (UUID)

An identifier used by DECrpc to identify interfaces, objects, and types.

unmarshall

To copy data from a Remote Procedure Call (RPC) packet. Stubs perform unmarshalling. *See also* **marshall.**

well known port

A port whose port number is part of the definition of an interface. Clients of the interface always send to that port; servers always listen on that port. *See also* **opaque port**.

Index

A

B

C

L

M

N

Q

R

S

U

V

W

X

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

DECrpc
Programming Guide
AA-PC2AA-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? ______________________________

____

What do you like best about this manual? ______________________________

____

What do you like least about this manual? ______________________________

____

Please list errors you have found in this manual:

Page   Description

______   ____

______   ____

______   ____

______   ____

Additional comments or suggestions to improve this manual:

____

____

____

What version of the software described by this manual are you using? __________

Name/Title ______________________________ Dept. __________

Company ______________________________ Date __________

Mailing Address ____

______________ Email ______________ Phone __________

---- Do Not Tear – Fold Here and Tape ----

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

Digital Equipment Corporation
Publications Manager
Open Software Publications Group
ZK03–2/Z04
110 SPIT BROOK ROAD
NASHUA, NH 03062–9987

--- Do Not Tear – Fold Here ---

Cut Along Dotted Line

Reader's Comments

DECrpc
Programming Guide
AA-PC2AA-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? ______________________________

What do you like best about this manual? ______________________________

What do you like least about this manual? ______________________________

Please list errors you have found in this manual:

Page Description

Additional comments or suggestions to improve this manual:

What version of the software described by this manual are you using? __________

Name/Title ______________________ Dept. __________

Company ______________________ Date __________

Mailing Address ______________________________

______________ Email ______________ Phone __________

---- Do Not Tear – Fold Here and Tape ----

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

Digital Equipment Corporation
Publications Manager
Open Software Publications Group
ZK03–2/Z04
110 SPIT BROOK ROAD
NASHUA, NH 03062–9987

--- Do Not Tear – Fold Here ---

Cut
Along
Dotted
Line