IONA®

Orbix®

PL/I Programmer's Guide and Reference

Version 6.0, November 2003

Making Software Work Together™

Updated: 14-Apr-2005

M 3 1 6 4

Contents

Part 3 Appendices

List of Figures

List of Tables

Preface

Orbix is a full implementation from IONA Technologies of the Common Object Request Broker Architecture (CORBA), as specified by the Object Management Group (OMG). Orbix complies with the following specifications:

- CORBA 2.3
- GIOP 1.2 (default), 1.1, and 1.0

Orbix Mainframe is IONA's implementation of the CORBA standard for the OS/390 platform. Orbix Mainframe documentation is periodically updated. New versions between release are available at http://www.iona.com/support/docs.

If you need help with this or any other IONA products, contact IONA at `support@iona.com`. Comments on IONA documentation can be sent to `docs-support@iona.com`.

Audience

This guide is intended for PL/I application programmers who want to develop Orbix applications in a native OS/390 environment.

Supported compilers

The supported compilers are:

- IBM PL/I for MVS & VM V1R1M1.
- IBM Enterprise PL/I for z/OS V3R2.

Organization of this guide

This guide is divided as follows:

Part 1, Programmer's Guide

Chapter 1, Introduction to Orbix

With Orbix, you can develop and deploy large-scale enterprise-wide CORBA systems in languages such as PL/I, COBOL, C++, and Java. Orbix has an advanced modular architecture that lets you configure and change functionality without modifying your application code, and a rich deployment architecture that lets you configure and manage a complex distributed system. Orbix Mainframe is IONA's CORBA solution for the OS/390 environment.

Chapter 2, Getting Started in Batch

This chapter introduces batch application programming with Orbix, by showing how to use Orbix to develop a simple distributed application that features a PL/I client and server, each running in batch.

Chapter 3, Getting Started in IMS

This chapter introduces IMS application programming with Orbix, by showing how to use Orbix to develop both an IMS PL/I client and an IMS PL/I server. It also provides details of how to subsequently run the IMS client against a PL/I batch server, and how to run a PL/I batch client against the IMS server.

Chapter 4, Getting Started in CICS

This chapter introduces CICS application programming with Orbix, by showing how to use Orbix to develop both a CICS PL/I client and a CICS PL/I server. It also provides details of how to subsequently run the CICS client against a PL/I batch server, and how to run a PL/I batch client against the CICS server.

Chapter 5, IDL Interfaces

The CORBA Interface Definition Language (IDL) is used to describe the interfaces of objects in an enterprise application. An object's interface describes that object to potential clients through its attributes and operations, and their signatures. This chapter describes IDL semantics and uses.

Chapter 6, IDL-to-PL/I Mapping

The CORBA Interface Definition Language (IDL) is used to define interfaces that are exposed by servers in your network. This chapter describes the standard IDL-to-PL/I mapping rules and shows, by example, how each IDL type is represented in PL/I.

Chapter 7, Orbix IDL Compiler

This chapter describes the Orbix IDL compiler in terms of the JCL used to run it, the PL/I members that it creates, the arguments that you can use with it, and the configuration settings that it uses.

Chapter 8, Memory Handling

Memory handling must be performed when using dynamic structures such as unbounded strings, unbounded sequences, and anys. This chapter provides details of responsibility for the allocation and subsequent release of dynamic memory for these complex types at the various stages of an Orbix PL/I application. It first describes in detail the memory handling rules adopted by the PL/I runtime for operation parameters relating to different dynamic structures. It then provides a type-specific breakdown of the APIs that are used to allocate and release memory for these dynamic structures.

Part 2, Programmer's Reference

Chapter 9, API Reference

This chapter summarizes the API functions that are defined for the Orbix PL/I runtime, in pseudo-code. It explains how to use each function, with an example of how to call it from PL/I.

Part 3, Appendices

Appendix A, POA Policies

This appendix summarizes the POA policies that are supported by the Orbix PL/I runtime, and the argument used with each policy.

Appendix B, System Exceptions

This appendix summarizes the Orbix system exceptions that are specific to the Orbix PL/I runtime.

Appendix C, Installed Data Sets

This appendix provides an overview listing of the data sets installed with Orbix Mainframe that are relevant to development and deployment of PL/I applications.

Related documentation

The document set for Orbix Mainframe includes the following related documentation:

- The *COBOL Programmer's Guide and Reference*, which provides details about developing, in a native OS/390 environment, Orbix COBOL applications that can run in batch, CICS, or IMS.
- The *CORBA Programmer's Guide, C++* and the *CORBA Programmer's Reference, C++*, which provide details about developing Orbix applications in C++ in various environments, including OS/390.
- The *Mainframe Migration Guide*, which provides details of migration issues for users who have migrated from IONA's Orbix 2.3-based solution for OS/390 to Orbix Mainframe.

The latest updates to the Orbix Mainframe documentation can be found at `http://www.iona.com/support/docs/orbix/6.0/mainframe/index.xml`.

Additional resources

The IONA knowledge base contains helpful articles, written by IONA experts, about Orbix and other products. You can access the knowledge base at the following location:

`http://www.iona.com/support/kb/`

The IONA update center contains the latest releases and patches for IONA products:

`http://www.iona.com/support/update/`

Typographical conventions

This guide uses the following typographical conventions:

`Constant width` — Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class.

Constant width paragraphs represent code examples or information a system displays on the screen. For example:

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

Italic	Italic words in normal text represent *emphasis* and *new terms*. Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/`*`your_name`* **Note:** Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

Keying conventions

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
`%`	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
`#`	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
`>`	The notation > represents the DOS, Windows NT, Windows 95, or Windows 98 command prompt.

`...`
`.`
`.`
`.`
	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.

`[ ]`	Brackets enclose optional items in format and syntax descriptions.
`{ }`	Braces enclose a list from which you must choose an item in format and syntax descriptions.
`|`	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

Part 1

Programmer's Guide

In this part

This part contains the following chapters:

Introduction to Orbix

With Orbix, you can develop and deploy large-scale enterprise-wide CORBA systems in languages such as PL/I, COBOL, C++, and Java. Orbix has an advanced modular architecture that lets you configure and change functionality without modifying your application code, and a rich deployment architecture that lets you configure and manage a complex distributed system. Orbix Mainframe is IONA's CORBA solution for the OS/390 environment.

In this chapter

This chapter discusses the following topics:

Why CORBA?

Need for open systems

Today's enterprises need flexible, open information systems. Most enterprises must cope with a wide range of technologies, operating systems, hardware platforms, and programming languages. Each of these is good at some important business task; all of them must work together for the business to function.

The common object request broker architecture—CORBA—provides the foundation for flexible and open systems. It underlies some of the Internet's most successful e-business sites, and some of the world's most complex and demanding enterprise information systems.

Need for high-performance systems

Orbix is a CORBA development platform for building high-performance systems. Its modular architecture supports the most demanding needs for scalability, performance, and deployment flexibility. The Orbix architecture is also language-independent, so you can implement Orbix applications in PL/I, COBOL, C++, or Java that interoperate via the standard IIOP protocol with applications built on any CORBA-compliant technology.

Open standard solution

CORBA is an open, standard solution for distributed object systems. You can use CORBA to describe your enterprise system in object-oriented terms, regardless of the platforms and technologies used to implement its different parts. CORBA objects communicate directly across a network using standard protocols, regardless of the programming languages used to create objects or the operating systems and platforms on which the objects run.

Widely available solution

CORBA solutions are available for every common environment and are used to integrate applications written in C, C++, Java, Ada, Smalltalk, COBOL, and PL/I running on embedded systems, PCs, UNIX hosts, and mainframes. CORBA objects running in these environments can cooperate seamlessly. Through OrbixCOMet, IONA's dynamic bridge between CORBA and COM, they can also interoperate with COM objects. CORBA offers an extensive infrastructure that supports all the features required by distributed business objects. This infrastructure includes important distributed services, such as transactions, messaging, and security.

CORBA Objects

Nature of abstract CORBA objects

CORBA objects are abstract objects in a CORBA system that provide distributed object capability between applications in a network. Figure 1 shows that any part of a CORBA system can refer to the abstract CORBA object, but the object is only implemented in one place and time on some server of the system.

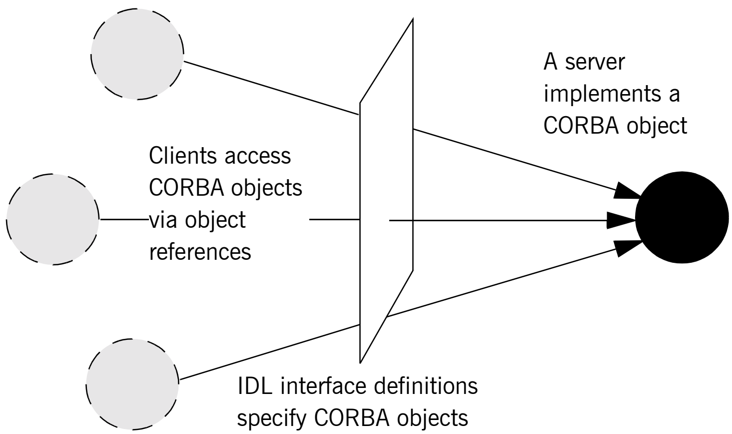

Figure 1: *The Nature of Abstract CORBA Objects*

Object references

An *object reference* is used to identify, locate, and address a CORBA object. Clients use an object reference to invoke requests on a CORBA object. CORBA objects can be implemented by servers in any supported programming language, such as PL/I, COBOL, C++, or Java.

IDL interfaces

Although CORBA objects are implemented using standard programming languages, each CORBA object has a clearly-defined interface, specified in the *CORBA Interface Definition Language (IDL)*. The *interface definition* specifies which member functions, data types, attributes, and exceptions are available to a client, without making any assumptions about an object's implementation.

Advantages of IDL

With a few calls to an ORB's application programming interface (API), servers can make CORBA objects available to client programs in your network.

To call member functions on a CORBA object, a client programmer needs only to refer to the object's interface definition. Clients use their normal programming language syntax to call the member functions of a CORBA object. A client does not need to know which programming language implements the object, the object's location on the network, or the operating system in which the object exists.

Using an IDL interface to separate an object's use from its implementation has several advantages. For example, you can change the programming language in which an object is implemented without affecting the clients that access the object. You can also make existing objects available across a network.

Object Request Broker

Overview

CORBA defines a standard architecture for object request brokers (ORB). An ORB is a software component that mediates the transfer of messages from a program to an object located on a remote network host. The ORB hides the underlying complexity of network communications from the programmer.

Role of an ORB

An ORB lets you create standard software objects whose member functions can be invoked by *client* programs located anywhere in your network. A program that contains instances of CORBA objects is often known as a *server*. However, the same program can serve at different times as a client and a server. For example, a server program might itself invoke calls on other server programs, and so relate to them as a client.

When a client invokes a member function on a CORBA object, the ORB intercepts the function call. As shown in Figure 2, the ORB redirects the function call across the network to the target object. The ORB then collects results from the function call and returns these to the client.

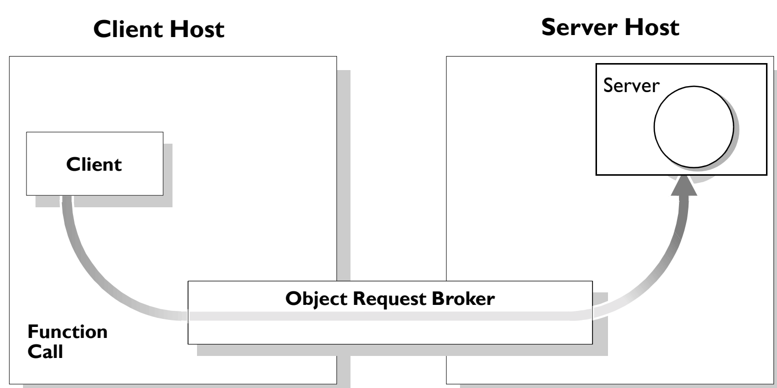

Figure 2: *The Object Request Broker*

CORBA Application Basics

Developing application interfaces

You start developing a CORBA application by defining interfaces to objects in your system in CORBA IDL. You compile these interfaces with an IDL compiler. An IDL compiler can generate PL/I, COBOL, C++, or Java from IDL definitions. Generated PL/I and COBOL consists of server skeleton code, which you use to implement CORBA objects.

Client invocations on CORBA objects

When an Orbix PL/I client on OS/390 calls a member function on a CORBA object on another platform, the call is transferred through the PL/I runtime to the ORB. (The client invokes on object references that it obtains from the server process.) The ORB then passes the function call to the server.

When a CORBA client on another platform calls a member function on an Orbix PL/I server object on OS/390, the ORB passes the function call through the PL/I runtime and then through the server skeleton code to the target object.

Orbix Plug-In Design

Overview

Orbix has a modular *plug-in* architecture. The ORB core supports abstract CORBA types and provides a plug-in framework. Support for concrete features like specific network protocols, encryption mechanisms, and database storage is packaged into plug-ins that can be loaded into the ORB, based on runtime configuration settings.

Plug-ins

A plug-in is a code library that can be loaded into an Orbix application at runtime. A plug-in can contain any type of code; typically, it contains objects that register themselves with the ORB runtimes to add functionality.

Plug-ins can be linked directly with an application, loaded when an application starts up, or loaded on-demand while the application is running. This gives you the flexibility to choose precisely those ORB features that you actually need. Moreover, you can develop new features such as protocol support for direct ATM or HTTPNG. Because ORB features are *configured* into the application rather than *compiled* in, you can change your choices as your needs change without rewriting or recompiling applications.

For example, an application that uses the standard IIOP protocol can be reconfigured to use the secure SSL protocol simply by configuring a different transport plug-in. There is no particular transport inherent to the ORB core; you simply load the transport set that suits your application best. This architecture makes it easy for IONA to support additional transports in the future such as multicast or special purpose network protocols.

ORB core

The ORB core presents a uniform programming interface to the developer: *everything is a CORBA object*. This means that everything appears to be a local PL/I, COBOL, C++, or Java object within the process, depending on which language you are using. In fact it might be a local object, or a remote object reached by some network protocol. It is the ORB's job to get application requests to the right objects no matter where they are located.

To do its job, the ORB loads a collection of plug-ins as specified by ORB configuration settings—either on startup or on demand—as they are needed by the application. For remote objects, the ORB intercepts local function calls and turns them into CORBA *requests* that can be dispatched to a remote object across the network via the standard IIOP protocol.

Orbix Application Deployment

Overview

Orbix provides a rich deployment environment designed for high scalability. You can create a *location domain* that spans any number of hosts across a network, and can be dynamically extended with new hosts. Centralized domain management allows servers and their objects to move among hosts within the domain without disturbing clients that use those objects. Orbix supports load balancing across object groups. A *configuration domain* provides the central control of configuration for an entire distributed application.

Orbix offers a rich deployment environment that lets you structure and control enterprise-wide distributed applications. Orbix provides central control of all applications within a common domain.

In this section

This section discusses the following topics:

Location Domains

Overview

A location domain is a collection of servers under the control of a single locator daemon. An Orbix location domain consists of two components: a *locator daemon* and a *node daemon*.

Note: See the *CORBA Administrator's Guide* for more details about these.

Locator daemon

The locator daemon can manage servers on any number of hosts across a network. The locator daemon automatically activates remote servers through a stateless activator daemon that runs on the remote host.

The locator daemon also maintains the implementation repository, which is a database of available servers. The implementation repository keeps track of the servers available in a system and the hosts they run on. It also provides a central forwarding point for client requests. By combining these two functions, the locator lets you relocate servers from one host to another without disrupting client request processing. The locator redirects requests to the new location and transparently reconnects clients to the new server instance. Moving a server does not require updates to the naming service, trading service, or any other repository of object references.

The locator can monitor the state of health of servers and redirect clients in the event of a failure, or spread client load by redirecting clients to one of a group of servers.

Node daemon

The node daemon acts as the control point for a single machine in the system. Every machine that will run an application server must be running a node daemon. The node daemon starts, monitors, and manages the application servers running on that machine. The locator daemon relies on the node daemons to start processes and inform it when new processes have become available.

Configuration Domains

Overview

A configuration domain is a collection of applications under common administrative control. A configuration domain can contain multiple location domains. During development, or for small-scale deployment, configuration can be stored in an ASCII text file, which is edited directly.

Plug-in design

The configuration mechanism is loaded as a plug-in, so future configuration systems can be extended to load configuration from any source such as example HTTP or third-party configuration systems.

Getting Started in Batch

This chapter introduces batch application programming with Orbix, by showing how to use Orbix to develop a simple distributed application that features a PL/I client and server, each running in its own region.

In this chapter

This chapter discusses the following topics:

Note: The example provided in this chapter does not reflect a real-world scenario that requires the Orbix Mainframe, because the supplied client and server are written in PL/I and running on OS/390. The example is supplied to help you quickly familiarize with the concepts of developing a batch PL/I application with Orbix.

Overview and Setup Requirements

Introduction

This section provides an overview of the main steps involved in creating an Orbix PL/I application. It describes important steps that you must perform before you begin. It also introduces the supplied `SIMPLE` demonstration, and outlines where you can find the various source code and JCL elements for it.

Steps to create an application

The main steps to create an Orbix PL/I application are:

Step	Action
1	"Developing the Application Interfaces" on page 21.
2	"Developing the Server" on page 28.
3	"Developing the Client" on page 36.

This chapter describes in detail how to perform each of these steps.

The Simple demonstration

This chapter describes how to develop a simple client-server application that consists of:

- An Orbix PL/I server that implements a simple persistent POA-based server.
- An Orbix PL/I client that uses the clearly defined object interface, `SimpleObject`, to communicate with the server.

The client and server use the Internet Inter-ORB Protocol (IIOP), which runs over TCP/IP, to communicate. As already stated, the `SIMPLE` demonstration is not meant to reflect a real-world scenario requiring the Orbix Mainframe, because the client and server are written in the same language and running on the same platform.

The Demonstration Server

The server accepts and processes requests from the client across the network. It is a batch server that runs in its own region.

See "Location of supplied code and JCL" for details of where you can find an example of the supplied server. See "Developing the Server" on page 28 for more details of how to develop the server.

The demonstration client

The client runs in its own region and accesses and requests data from the server. When the client invokes a remote operation, a request message is sent from the client to the server. When the operation has completed, a reply message is sent back to the client. This completes a single remote CORBA invocation.

See "Location of supplied code and JCL" for details of where you can find an example of the supplied client. See "Developing the Client" on page 36 for more details of how to develop the client.

Location of supplied code and JCL

All the source code and JCL components needed to create and run the batch `SIMPLE` demonstration have been provided with your installation. Apart from site-specific changes to some JCL, these do not require editing.

Table 1 provides a summary of the supplied code elements and JCL components that are relevant to the batch `SIMPLE` demonstration (where *orbixhlq* represents your installation's high-level qualifier).

Table 1: *Supplied Code and JCL (Sheet 1 of 2)*

Location	Description
orbixhlq.DEMOS.IDL(SIMPLE)	This is the supplied IDL.
orbixhlq.DEMOS.PLI.SRC(SIMPLEV)	This is the source code for the batch server mainline module.
orbixhlq.DEMOS.PLI.SRC(SIMPLEI)	This is the source code for the batch server implementation module.
orbixhlq.DEMOS.PLI.SRC(SIMPLEC)	This is the source code for the client module.
orbixhlq.JCL(LOCATOR)	This JCL runs the Orbix locator daemon.
orbixhlq.JCL(NODEDAEM)	This JCL runs the Orbix node daemon.

Table 1: *Supplied Code and JCL (Sheet 2 of 2)*

Location	Description
orbixhlq.DEMOS.PLI.BLD.JCL(SIMPLIDL)	This JCL runs the Orbix IDL compiler, to generate PL/I source and include members for the batch server. This JCL specifies the `-V` compiler argument, which stops generation of server mainline code by default. The `-S` compiler argument, which generates server implementation code, is disabled by default in this JCL.
orbixhlq.DEMOS.PLI.BLD.JCL(SIMPLECB)	This JCL compiles the client module to create the `SIMPLE` client program.
orbixhlq.DEMOS.PLI.BLD.JCL(SIMPLESB)	This JCL compiles and links the batch server mainline and implementation modules to create the `SIMPLE` server program.
orbixhlq.DEMOS.PLI.RUN.JCL(SIMPLESV)	This JCL runs the server.
orbixhlq.DEMOS.PLI.BLD.JCL(SIMPLECL)	This JCL runs the client.

Note: Other code elements and JCL components are provided for the IMS and CICS versions of the `SIMPLE` demonstration. See "Getting Started in IMS" on page 49 and "Getting Started in CICS" on page 93 for more details of these.

Supplied include members

Table 2 provides a summary in alphabetic order of the various include members that are supplied with your product installation. In Table 2, *servers* means batch servers, and *clients* means batch clients. Again, *orbixhlq* represents your installation's high-level qualifier.

Table 2: *Supplied Include Members (Sheet 1 of 3)*

Location	Description
orbixhlq.INCLUDE.PLINCL(CHKERRS)	This contains a PL/I function that can be called both by clients and servers to check if a system exception has occurred, and to report that system exception.

Table 2: *Supplied Include Members (Sheet 2 of 3)*

Location	Description
orbixhlq.INCLUDE.PLINCL(CORBA)	This contains common PL/I runtime variables that can be used both by clients and servers. It includes the CORBACOM include member by default. It also includes the CORBASV include member, if the client program contains the line %client_only='yes';.
orbixhlq.INCLUDE.PLINCL(CORBACOM)	This contains common PL/I runtime function definitions that can be used both by clients and servers.
orbixhlq.INCLUDE.PLINCL(CORBASV)	This contains PL/I runtime function definitions that can be used by servers.
orbixhlq.INCLUDE.PLINCL(DISPINIT)	This is used by servers. It retrieves the current request information into the REQINFO structure via PODREQ. From REQINFO the operation to be performed by the server is retrieved via a call to STRGET.
orbixhlq.INCLUDE.PLINCL(EXCNAME)	This is relevant to both batch clients and servers. It contains a PL/I function called CORBA_EXC_NAME that returns the system exception name for the system exception being raised (that is, it maps Orbix exceptions to human-readable strings). EXCNAME is used by CHKERRS.
orbixhlq.INCLUDE.PLINCL(IORREC)	This is used both by clients and servers. It contains declarations for the IOR file and associated variables.
orbixhlq.INCLUDE.PLINCL(READIOR)	This is used both by clients and servers. It reads the IOR from IORFILE, and converts the PL/I character string that is read into an unbounded string. This string is subsequently used by the OBJ2STR function, to create an object reference from the IOR that has been read.
orbixhlq.INCLUDE.PLINCL(SETUPCL)	This is used by clients. It sets up several ON…ERROR blocks that check the status of IORFILE and catch any general errors that might occur in the client.

Table 2: *Supplied Include Members (Sheet 3 of 3)*

Location	Description
orbixhlq.INCLUDE.PLINCL(URLSTR)	This is relevant to clients only. It contains a PL/I representation of the corbaloc URL IIOP string format. A client can call `STR2OBJ` to convert the URL into an object reference. See “STR2OBJ” on page 404 for more details. **Note:** Even though batch applications can use this include member, the supplied batch demonstration does not use this.
orbixhlq.DEMOS.PLI.PLINCL	This PDS is used to store all batch include members that are generated by the Orbix IDL compiler when you run the supplied `SIMPLIDL` JCL for the batch demonstration. It also contains helper procedures for the bank, naming, and nested sequences demonstrations.

Note: An `IMSPCB` include member is also provided specifically for use with IMS servers. See “Getting Started in IMS” on page 49 for more details of this.

Checking JCL components

When creating the simple application, check that each step involved within the separate JCL components completes with a condition code not greater than `4`. If the condition codes are greater than `4`, establish the point and cause of failure. The most likely cause is the site-specific JCL changes required for the compilers. Ensure that each high-level qualifier throughout the JCL reflects your installation.

Developing the Application Interfaces

Overview

This section describes the steps you must follow to develop the IDL interfaces for your application. It first describes how to define the IDL interfaces for the objects in your system. It then describes how to generate PL/I source and include members from IDL interfaces, and provides a description of the members generated from the supplied `SimpleObject` interface.

Steps to develop application interfaces

The steps to develop the interfaces to your application are:

Step	Action
1	Define public IDL interfaces to the objects required in your system. See "Defining IDL Interfaces" on page 22.
2	Use the Orbix IDL compiler to generate PL/I source and include members from the defined IDL. See "Generating PL/I Source and Include Members" on page 23.

Defining IDL Interfaces

Defining the IDL

The first step in writing an Orbix program is to define the IDL interfaces for the objects required in your system. The following is an example of the IDL for the `SimpleObject` interface that is supplied in *orbixhlq*.DEMOS.IDL(SIMPLE):

```
// IDL
module Simple
{
    interface SimpleObject
    {
        void
        call_me();
    };
};
```

Explanation of the IDL

The preceding IDL declares a `SimpleObject` interface that is scoped (that is, contained) within the `Simple` module. This interface exposes a single `call_me()` operation. This IDL definition provides a language-neutral interface to the CORBA `Simple::SimpleObject` type.

How the demonstration uses this IDL

For the purposes of this example, the `SimpleObject` CORBA object is implemented in PL/I in the supplied `Simple` server application. The server application creates a persistent server object of the `SimpleObject` type, and publishes its object reference to a PDS member. The client application must then locate the `SimpleObject` object by reading the IOR from the relevant PDS member. The client invokes the `call_me()` operation on the `SimpleObject` object, and then exits.

Generating PL/I Source and Include Members

The Orbix IDL compiler

You can use the Orbix IDL compiler to generate PL/I source and include members from IDL definitions.

Orbix IDL compiler configuration

The Orbix IDL compiler uses the Orbix configuration member for its settings. The `SIMPLIDL` JCL that runs the compiler uses the configuration member *orbixhlq*`.CONFIG(IDL)`. See "Orbix IDL Compiler Configuration" on page 263 for more details of this configuration member.

Running the Orbix IDL compiler

The PL/I source for the batch server demonstration described in this chapter is generated in the first step of the following job:

```
orbixhlq.DEMOS.PLI.BLD.JCL(SIMPLIDL)
```

Generated source code members

Table 3 shows the server source code members that the Orbix IDL compiler generates, based on the defined IDL:

Table 3: *Generated Server Source Code Members*

Member	JCL Keyword Parameter	Description
idlmembername`I`	`IMPL`	This is the server implementation source code member. It contains procedure definitions for all the callable operations. The is only generated if you specify the `-S` argument with the IDL compiler.
idlmembername`V`	`IMPL`	This is the server mainline source code member. It is generated by default. However, you can use the `-V` argument with the IDL compiler, to prevent generation of this member.

Note: For the purposes of this example, the `SIMPLEI` server implementation and `SIMPLEV` server mainline are already provided in your product installation. Therefore, the `-S` argument, which generates server implementation code, is not specified in the supplied `SIMPLIDL` JCL. The `-V` argument, which prevents generation of server mainline code, is specified in the supplied JCL. See "Orbix IDL Compiler" on page 233 for more details of the IDL compiler arguments used to generate, and prevent generation of, server source code.

Generated PL/I include members

Table 4 shows the PL/I include members that the Orbix IDL compiler generates, based on the defined IDL.

Table 4: *Generated PL/I Include Members*

Copybook	JCL Keyword Parameter	Description
*idlmembername*D	COPYLIB	This include member contains a select statement that determines which server implementation procedure is to be called, based on the interface name and operation received.
*idlmembername*L	COPYLIB	This include member contains structures and procedures used by the PL/I runtime to read and store data into the operation parameters. This member is automatically included in the *idlmembername*X include member.
*idlmembername*M	COPYLIB	This include member contains declarations and structures that are used for working with operation parameters and return values for each interface defined in the IDL member. The structures use the based PL/I structures declared in the *idlmembername*T include member. This member is automatically included in the *idlmembername*I include member.

Table 4: *Generated PL/I Include Members*

Copybook	JCL Keyword Parameter	Description
*idlmembername*T	COPYLIB	This include member contains the based structure declarations that are used in the *idlmembername*M include member. This member is automatically included in the *idlmembername*M include member.
*idlmembername*X	COPYLIB	This include member contains structures that are used by the PL/I runtime to support the interfaces defined in the IDL member. This member is automatically included in the *idlmembername*V source code member.
*idlmembername*D	COPYLIB	This include member contains a select statement for calling the correct procedure for the requested operation. This include member is automatically included in the *idlmembername*I source code member.

How IDL maps to PL/I include members

Each IDL interface maps to a set of PL/I structures. There is one structure defined for each IDL operation. A structure contains each of the parameters for the relevant IDL operation in their corresponding PL/I representation. See "IDL-to-PL/I Mapping" on page 177 for details of how IDL types map to PL/I.

Attributes map to two operations (`get` and `set`), and readonly attributes map to a single `get` operation.

Member name restrictions

Generated PL/I source code and include member names are all based on the IDL member name. If the IDL member name exceeds six characters, the Orbix IDL compiler uses only the first six characters of the IDL member name when generating include member names. This allows space for appending a one-character suffix to each generated member name, while allowing it to adhere to the seven-character maximum size limit for PL/I external procedure names, which are based by default on the generated member names.

Location of demonstration include members

You can find examples of the include members generated for the `Simple` demonstration in the following locations:

- *orbixhlq*.DEMOS.PLI.PLINCL(SIMPLED)
- *orbixhlq*.DEMOS.PLI.PLINCL(SIMPLEL)
- *orbixhlq*.DEMOS.PLI.PLINCL(SIMPLEM)
- *orbixhlq*.DEMOS.PLI.PLINCL(SIMPLET)
- *orbixhlq*.DEMOS.PLI.PLINCL(SIMPLEX)

Note: These include members are not shipped with your product installation. They are generated when you run the supplied `SIMPLIDL` JCL, to run the Orbix IDL compiler.

Developing the Server

Overview

This section describes the steps you must follow to develop the batch server executable for your application.

Steps to develop the server

The steps to develop the server application are:

Step	Action
1	"Writing the Server Implementation" on page 29
2	"Writing the Server Mainline" on page 32
3	"Building the Server" on page 35.

Writing the Server Implementation

The server implementation module

You must complete the server implementation by writing the logic that implements each operation in the `idlmembernameI` source code member. For the purposes of this example, you must write a PL/I procedure that implements each operation in the `SIMPLEI` member.

When you specify the `-S` argument with the Orbix IDL compiler in this case, it generates a skeleton module called `SIMPLEI`, which generates an empty procedure for each attribute and operation within the interface.

Example of the completed SIMPLEI module

The following is an example of the completed `SIMPLEI` module (with the header comment block omitted for the sake of brevity):

Example 1: *The SIMPLEI Demonstration Module (Sheet 1 of 2)*

```
   SIMPLEI: PROC;

   /*The following line enables the runtime to call this procedure*/
1  DISPTCH: ENTRY;

   dcl (addr,low,sysnull)                  builtin;

   %include CORBA;
   %include CHKERRS;
   %include SIMPLEM;
2  %include DISPINIT;

   /* ================ Start of global user code =============== */
   /* ================= End of global user code ================ */

   /* ------------------------------------------------------------*/
   /*                                                            */
   /* Dispatcher : select(operation)                             */
   /*                                                            */
   /*------------------------------------------------------------*/
3  %include SIMPLED;


   /*------------------------------------------------------------*/
   /* Interface:                                                 */
   /*     Simple/SimpleObject                                    */
   /*                                                            */
```

Example 1: *The SIMPLEI Demonstration Module (Sheet 2 of 2)*

```
  /* Mapped name:                                               */
  /*     Simple_SimpleObject                                    */
  /*                                                            */
  /* Inherits interfaces:                                       */
  /*     (none)                                                 */
  /*------------------------------------------------------------*/
  /*------------------------------------------------------------*/
  /* Operation:         call_me                                 */
  /* Mapped name:       call_me                                 */
  /* Arguments:         None                                    */
  /* Returns:           void                                    */
  /*------------------------------------------------------------*/
4 proc_Simple_SimpleObject_c_c904: PROC(p_args);

  dcl p_args                    ptr;
5 dcl 1_args                    aligned based(p_args)
                              like Simple_SimpleObject_c_ba77_type;

  /* ==================== Start of operation code ============= */
6 put skip list('Operation call_me() called');
  put skip;
  /* ===================== End of operation code ============== */

  END proc_Simple_SimpleObject_c_c904;

  END SIMPLEI;
```

Explanation of the SIMPLEI module

The SIMPLEI module can be explained as follows:

1. When an incoming request arrives from the network, it is processed by the ORB and a call is made from the PL/I runtime to the DISPTCH entry point.
2. Within the DISPINIT include member, PODREQ is called to provide information about the current invocation request, which is held in the REQINFO structure. PODREQ is called once for each operation invocation after a request has been dispatched to the server. STRGET is then called to copy the characters in the unbounded string pointer for the operation name into the PL/I string that represents the operation name.
3. The SIMPLED include member contains a select statement that determines which procedure within SIMPLEI is to be called, given the operation name and interface name passed to SIMPLEI. It calls PODGET

before the call to the server procedure, which fills the appropriate PL/I structure declared in the main include member, `SIMPLEM`, with the operation's incoming arguments. It then calls `PODPUT` after the call to the server procedure, to send out the operation's outgoing arguments.

4. The procedural code containing the server implementation for the `call_me` operation.
5. Each operation has an argument structure and these are declared in the typecode include member, `SIMPLET`. If an operation does not have any parameters or return type, such as `call_me`, the structure only contains a structure with a dummy char.
6. This is a sample of the server implementation code for `call_me`. It is the only part of the `SIMPLEI` member that is not automatically generated by the Orbix IDL compiler.

Location of the SIMPLEI module

You can find a complete version of the `SIMPLEI` server implementation module in *`orbixhlq`*`.DEMOS.PLI.SRC(SIMPLEI)`.

Writing the Server Mainline

The server mainline module

The next step is to write the server mainline module in which to run the server implementation. The Orbix IDL compiler generates the server mainline module, `SIMPLEV`, by default. However, you can prevent generation of the server mainline module by specifying the `-V` argument with the IDL compiler. The `-V` argument therefore allows you to prevent overwriting any customized changes you might have already made to the server mainline.

Example of the SIMPLEV module

The following is an example of the `SIMPLEV` module (with the header comment block omitted for the sake of brevity):

Example 2: *The SIMPLEV Demonstration Module (Sheet 1 of 2)*

```
SIMPLEV: PROC OPTIONS(MAIN);

dcl arg_list                 char(01)       init(’’);
dcl arg_list_len             fixed bin(31)  init(0);
dcl orb_name                 char(10)       init(’simple_orb’);
dcl orb_name_len             fixed bin(31)  init(10);
dcl srv_name                 char(256) var;
dcl server_name              char(07)       init(’simple ’);
dcl server_name_len          fixed bin(31)  init(6);

dcl Simple_SimpleObject_obj  ptr;

dcl DISPTCH                    ext entry;
dcl IORFILE                    file record output;
dcl SYSPRINT                   file stream output;
dcl (addr,length,low,sysnull)  builtin;

%include CORBA;
%include CHKERRS;
%include IORREC;
%include SIMPLET;
%include SIMPLEX;

alloc pod_status_information set(pod_status_ptr);
1 call podstat(pod_status_ptr);
if check_errors(’podstat’) ^= completion_status_yes then return;

/* Initialize the server connection to the ORB                 */
```

Example 2: *The SIMPLEV Demonstration Module (Sheet 2 of 2)*

```
2   call orbargs(arg_list,arg_list_len,orb_name,orb_name_len);
    if check_errors('orbargs') ^= completion_status_yes then return;

3   call podsrvr(server_name, server_name_len);
    if check_errors('podsrvr') ^= completion_status_yes then return;

    /* Register interface : Simple/SimpleObject                       */
4   call podreg(addr(Simple_SimpleObject_interface));
    if check_errors('podreg';) ^= completion_status_yes then return;

    put skip list('Creating the simple_persistent object');
5   call objnew(server_name, Simple_SimpleObject_intf,
       Simple_SimpleObject_objid, Simple_SimpleObject_obj);
    if check_errors('objnew') ^= completion_status_yes then return;

    /* Write out the IOR for each interface                           */
    open file(IORFILE);

6   call obj2str(Simple_SimpleObject_obj, iorrec_ptr);
    if check_errors('obj2str') ^= completion_status_yes then return;

    put skip list('Writing out the object reference');
    call strget(iorrec_ptr, iorrec, iorrec_len);
    if check_errors('strget') ^= completion_status_yes then return;

    write file(IORFILE) from(iorrec);
    close file(IORFILE);

    /* Server is now ready to accept requests                         */
    put skip list('Giving control to the ORB to process requests');
    put skip;
7   call podrun;
    if check_errors('podrun') ^= completion_status_yes then return;

8   call objrel(Simple_SimpleObject_obj);
    if check_errors('objrel') ^= completion_status_yes then return;

    free pod_status_information;

    END SIMPLEV;
```

Explanation of the SIMPLEV module

The `SIMPLEV` module can be explained as follows:

1. `PODSTAT` is called to register the `POD_STATUS_INFORMATION` block that is contained in the `CORBA` include member. Registering the `POD_STATUS_INFORMATION` block allows the PL/I runtime to populate it with exception information, if necessary. If `completion_status` is set to zero after a call to the PL/I runtime, this means that the call has completed successfully.
2. `ORBARGS` is called to initialize a connection to the ORB.
3. `PODSRVR` is called to set the server name.
4. `PODREG` is called to register the IDL interface, `SimpleObject`, with the PL/I runtime.
5. `OBJNEW` is called to create a unique object reference from the server name, interface name, and object ID for the server.
6. `OBJ2STR` is called to translate the object reference created by `OBJNEW` into a stringified IOR. The stringified IOR is then written to the `IORFILE` member.
7. `PODRUN` is called, to enter the `ORB::run()` loop, to allow the ORB to receive and process client requests.
8. `OBJREL` is called to ensure that the servant object is released properly.

See the preface of this guide for details about the compilers that this product supports.

Location of the SIMPLEV module

You can find a complete version of the `SIMPLEV` server mainline in *orbixhlq*`.DEMOS.PLI.SRC(SIMPLEV)`.

Building the Server

Location of the JCL

Sample JCL used to compile and link the batch server mainline and server implementation is in *orbixhlq*.DEMOS.PLI.BLD.JCL(SIMPLESB).

Resulting load module

When this JCL has successfully executed, it results in a load module that is contained in *orbixhlq*.DEMOS.PLI.LOAD(SIMPLESV).

Server programming restrictions

Although the server implementation code is compiled as part of the main program, it effectively executes as a dynamically loaded procedure. The fetch and release restrictions documented in the IBM publication: *IBM PL/I for MVS* & *VM Language Reference Release 1.1: SC26-3114* must be observed. Failure to observe these restrictions can result in various errors, including S0C4, S22C, and U4094 abends.

For example, all files to be used by the server program must be explicitly opened before the first Orbix PL/I runtime call in the server mainline and must be explicitly closed at the end of the server mainline.

Developing the Client

Overview

This section describes the steps you must follow to develop the client executable for your application.

Note: The Orbix IDL compiler does not generate PL/I client stub code.

Steps to develop the client

The steps to develop the client application are:

Step	Action
1	“Writing the Client” on page 37.
2	“Building the Client” on page 40.

Writing the Client

The client program

The next step is to write the client program. This example uses the supplied `SIMPLEC` client demonstration.

Example of the SIMPLEC program

The following is an example of the `SIMPLEC` program

Example 3: *The SIMPLEC Demonstration Program (Sheet 1 of 2)*

```
SIMPLEC: PROC OPTIONS(MAIN);

1 %client_only='yes';

dcl (addr, null, substr, sysnull)  builtin;
dcl SYSIN                          file input;
dcl SYSPRINT                       file stream output;

dcl arg_list                  char(40)       init('');
dcl arg_list_len              fixed bin(31)  init(38);
dcl orb_name                  char(10)       init('simple_orb');
dcl orb_name_len              fixed bin(31)  init(10);

dcl Simple_SimpleObject_obj   ptr;

%include CORBA;
%include CHKERRS;
%include SIMPLEM;
%include SIMPLEX;

%include SETUPCL;               /* Various DCLs for the client */
%include IORREC;                /* Describes the IOR file type */

open file(IORFILE) input;       /* Open the server IOR member  */
2 %include READIOR;             /* Read in the server's IOR    */

/* General Client Setup  */
/* Initialize the PL/I runtime status information block  */
alloc pod_status_information set(pod_status_ptr);
call podstat(pod_status_ptr);

/* Initialize our ORB  */
3 call orbargs(arg_list, arg_list_len, orb_name, orb_name_len);
```

Example 3: *The SIMPLEC Demonstration Program (Sheet 2 of 2)*

```
  /* Register the SimpleObject interface with the PL/I runtime */
4 call podreg(addr(Simple_SimpleObject_interface));
  if check_errors('podreg') ^= completion_status_yes then return;

  /* Create an object reference from the server's IOR  */
  /* so we can make calls to the server                */
5 call str2obj(iorrec_ptr, Simple_SimpleObject_obj);
  if check_errors('objset') ^= completion_status_yes then return;

  /* Now we are ready to start making server requests  */
  put skip list('simple_persistent demo');
  put skip list('======================');

  /* Call operation call_me  */
  /* As this is a very simple function, there aren't any    */
  /* parameters. So instead we pass in the generated dummy  */
  /* structure created for this operation.                  */
  put skip list('Calling operation call_me…');
6 call podexec(Simple_SimpleObject_obj,
      Simple_SimpleObject_call_me,
      addr(Simple_SimpleObject_c_ba77_args),
      no_user_exceptions);
  if check_errors('podexec') ^= completion_status_yes then return;

  put skip list('Operation call_me completed (no results to
     display)');
  put skip;
  put skip list('End of the simple_persistent demo');
  put skip;

  /* Free the simple_persistent object reference */
7 call objrel(Simple_SimpleObject_obj);
  if check_errors('objrel') ^= completion_status_yest then return;

  END SIMPLEC;
```

Explanation of the SIMPLEC program

The `SIMPLEC` program can be explained as follows:

1. This preprocessor setting instructs the PL/I compiler not to include the `CORBASV` include member, which contains PL/I runtime functions that are used only by the server. The `CORBA` include member includes a check for this setting.
2. The `READIOR` include member reads the IOR from the `IORFILE` member and creates an unbounded string, called `iorrec_ptr`, which is used later in the program to create an object reference from this IOR.
3. `ORBARGS` is called to initialize a connection to the ORB.
4. `PODREG` is called to register the IDL interface with the PL/I runtime.
5. `STR2OBJ` is called to create an object reference to the server object represented by the IOR. This must be done to allow operation invocations on the server. The `STR2OBJ` call takes an interoperable stringified object reference and produces an object reference pointer. This pointer is used in all method invocations. See the *CORBA Programmer's Reference, C++* for more details about stringified object references.
6. After the object reference is created, `PODEXEC` is called to invoke operations on the server object represented by that object reference. You must pass the object reference, the operation name, the argument description packet, and the user exception buffer. If the call does not have a user exception defined (as in the preceding example), the `no_user_exceptions` variable is passed in instead. The operation name must have at least one trailing space. The same argument description is used by the server, and can be found in the *orbixhlq*`.DEMOS.PLI.PLINCL(SIMPLET)` include member.
7. `OBJREL` is called to ensure that the servant object is released properly.

Location of the SIMPLEC program

You can find a complete version of the `SIMPLEC` client module in *orbixhlq*`.DEMOS.PLI.SRC(SIMPLEC)`.

Building the Client

Location of the JCL

Sample JCL used to compile and link the client can be found in the third step of *orbixhlq*.DEMOS.PLI.BLD.JCL(SIMPLECB).

Resulting load module

When the JCL has successfully executed, it results in a load module that is contained in *orbixhlq*.DEMOS.PLI.LOAD(SIMPLECL).

Running the Application

Introduction

This section describes the steps you must follow to run your application. It also provides an example of the output produced by the client and server.

Note: This example involves running a PL/I client and PL/I server. You could, however, choose to run a PL/I server and a C++ client, or a PL/I client and a C++ server. Substitution of the appropriate JCL is all that is required in the following steps to mix clients and servers in different languages.

Steps to run the application

The steps to run the application are:

Step	Action
1	“Starting the Orbix Locator Daemon” on page 42 (if it has not already been started).
2	“Starting the Orbix Node Daemon” on page 43 (if it has not already been started).
3	“Running the Server and Client” on page 44.

Starting the Orbix Locator Daemon

Overview

An Orbix locator daemon must be running on the server's location domain before you try to run your application. The Orbix locator daemon is a program that implements several components of the ORB, including the Implementation Repository. The locator runs in its own address space on the server host, and provides services to the client and server, both of which need to communicate with it.

When you start the Orbix locator daemon, it appears as an active job waiting for requests. See the *CORBA Administrator's Guide* for more details about the locator daemon.

JCL to start the Orbix locator daemon

If the Orbix locator daemon is not already running, you can use the JCL in `orbixhlq.JCL(LOCATOR)` to start it.

Locator daemon configuration

The Orbix locator daemon uses the Orbix configuration member for its settings. The JCL that you use to start the locator daemon uses the configuration member `orbixhlq.CONFIG(DEFAULT@)`.

Starting the Orbix Node Daemon

Overview

An Orbix node daemon must be running on the server's location domain before you try to run your application. The node daemon acts as the control point for a single machine in the system. Every machine that will run an application server must be running a node daemon. The node daemon starts, monitors, and manages the application servers running on that machine. The locator daemon relies on the node daemons to start processes and inform it when new processes have become available.

When you start the Orbix node daemon, it appears as an active job waiting for requests. See the *CORBA Administrator's Guide* for more details about the node daemon.

JCL to start the Orbix node daemon

If the Orbix node daemon is not already running, you can use the JCL in `*orbixhlq*.JCL(NODEDAEM)` to start it.

Node daemon configuration

The Orbix node daemon uses the Orbix configuration member for its settings. The JCL that you use to start the node daemon uses the configuration member `*orbixhlq*.CONFIG(DEFAULT@)`.

Running the Server and Client

JCL to run the server

To run the supplied `SIMPLESV` server application, use the following JCL:

```
orbixhlq.DEMOS.PLI.JCL(SIMPLESV)
```

Note: You can use the OS/390 `STOP` operator command to stop the server.

IOR member for the server

When you run the server, it automatically writes its IOR to a PDS member that is subsequently used by the client. For the purposes of this example, the IOR member is contained in *`orbixhlq`*`.DEMOS.IORS(SIMPLE)`.

JCL to run the client

After you have started the server and made it available to the network, you can use the following JCL to run the supplied `SIMPLECL` client application:

```
orbixhlq.DEMOS.PLI.RUN.JCL(SIMPLECL)
```

Application Output

Server output

The following is an example of the output produced by the simple server:

```
Creating the simple_persistent object
Writing out the object reference
Giving control to the ORB to process Requests

Operation call_me() called
```

Client output

The following is an example of the output produced by the simple client:

```
simple_persistent demo
=======================
Calling operation call me…
Operation call_me completed (no results to display)

End of the simple_persistent demo
```

Result

If you receive the preceding client and server output, it means that you have successfully created an Orbix PL/I client-server batch application.

Application Address Space Layout

Overview

Figure 3 is a graphical overview of the address space layout for an Orbix PL/I application running in batch in a native OS/390 environment. This is shown for the purposes of example and is not meant to reflect a real-world scenario requiring the Orbix Mainframe.

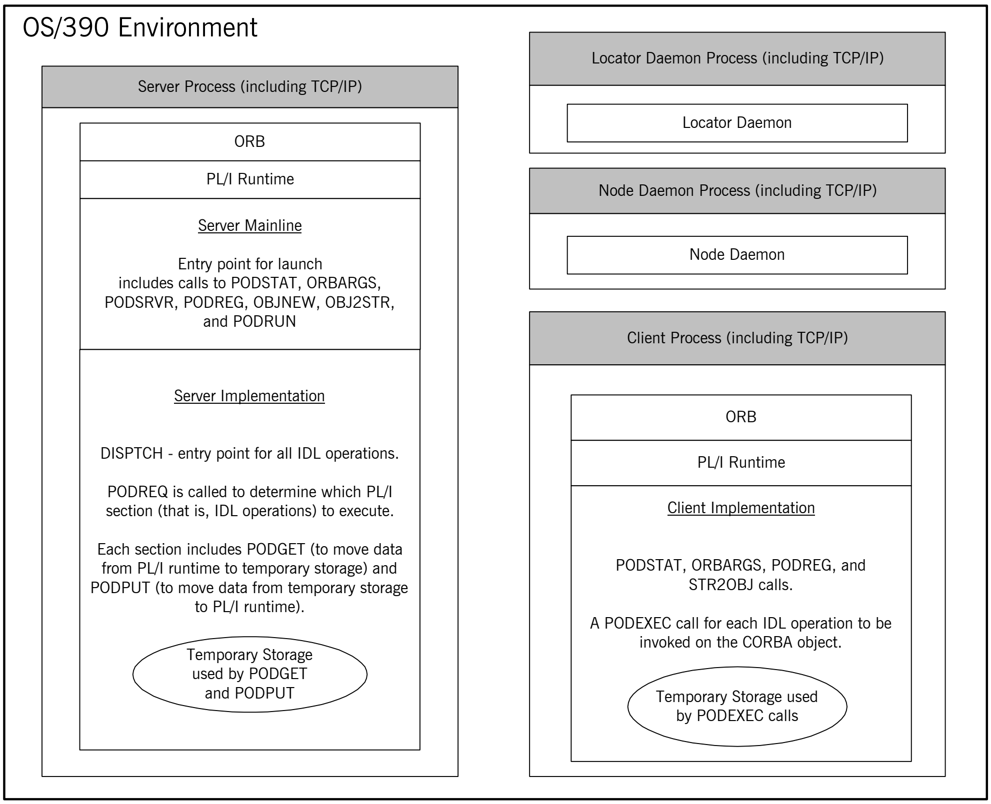

Figure 3: *Address Space Layout for an Orbix PL/I Application*

Explanation of the server process

The server-side ORB, PL/I runtime, server mainline (launch entry point), and server implementation are linked into a single load module referred to as the "server". The PL/I runtime marshals data to and from the server implementation’s operation structures, which means there is language-specific translation between C++ and PL/I.

The server runs within its own address space. It uses the TCP/IP protocol to communicate (through the server-side ORB) with both the client and the locator daemon.

For an example and details of:

- The APIs called by the server mainline, see “Explanation of the SIMPLEV module” on page 34 and “API Reference” on page 305.
- The APIs called by the server implementation, see “Explanation of the SIMPLEI module” on page 30 and “API Reference” on page 305.

Explanation of the daemon processes

The locator daemon and node daemon each runs in its own address space. See “Location Domains” on page 12 for more details of the locator and node daemons.

The locator daemon and node daemon use the TCP/IP protocol to communicate with each other. The locator daemon also uses the TCP/IP protocol to communicate with the server through the server-side ORB.

Explanation of the client process

The client-side ORB, PL/I runtime, and client implementation are linked into a single load module referred to as the “client”. The client runs within its own address space.

The client (through the client-side ORB) uses TCP/IP to communicate with the server.

For an example and details of the APIs called by the client, see “Explanation of the SIMPLEC program” on page 39 and “API Reference” on page 305.

Getting Started in IMS

This chapter introduces IMS application programming with Orbix, by showing how to use Orbix to develop both an IMS PL/I client and an IMS PL/I server. It also provides details of how to subsequently run the IMS client against a PL/I batch server, and how to run a PL/I batch client against the IMS server.

In this chapter

This chapter discusses the following topics:

Note: The client and server examples provided in this chapter respectively require use of the IMS client and server adapters that are supplied as part of the Orbix Mainframe. See the *IMS Adapters Administrator's Guide* for more details about these IMS adapters.

Overview

Introduction

This section provides an overview of the main steps involved in creating an Orbix PL/I IMS server and client application. It also introduces the supplied PL/I IMS client and server `SIMPLE` demonstrations, and outlines where you can find the various source code and JCL elements for them.

Steps to create an application

The main steps to create an Orbix PL/I IMS server application are:

1. “Developing the Application Interfaces” on page 56.
2. “Developing the IMS Server” on page 67.
3. “Developing the IMS Client” on page 79.

For the purposes of illustration this chapter demonstrates how to develop both an Orbix PL/I IMS client and an Orbix PL/I IMS server. It then describes how to run the IMS client and IMS server respectively against a PL/I batch server and a PL/I batch client. These demonstrations do not reflect real-world scenarios requiring the Orbix Mainframe, because the client and server are written in the same language and running on the same platform.

The demonstration IMS server

The Orbix PL/I server developed in this chapter runs in an IMS region. It implements a simple persistent POA-based obect. It accepts and processes requests from an Orbix PL/I batch client that uses the object interface, `SimpleObject`, to communicate with the server via the IMS server adapter. The IMS server uses the Internet Inter-ORB Protocol (IIOP), which runs over TCP/IP, to communicate with the batch client.

The demonstration IMS client

The Orbix PL/I client developed in this chapter runs in an IMS region. It uses the clearly defined object interface, `SimpleObject`, to access and request data from an Orbix PL/I batch server that implements a simple persistent `SimpleObject` object. When the client invokes a remote operation, a request message is sent from the client to the server via the client adapter. When the operation has completed, a reply message is sent back to the client again via the client adapter. The IMS client uses IIOP to communicate with the batch server.

Supplied code and JCL for IMS application development

All the source code and JCL components needed to create and run the IMS SIMPLE server and client demonstrations have been provided with your installation. Apart from site-specific changes to some JCL, these do not require editing.

Table 5 provides a summary of these code elements and JCL components (where *orbixhlq* represents your installation's high-level qualifier).

Table 5: *Supplied Code and JCL (Sheet 1 of 2)*

Location	Description
orbixhlq.DEMOS.IDL(SIMPLE)	This is the supplied IDL.
orbixhlq.DEMOS.IMS.PLI.SRC (SIMPLESV)	This is the source code for the IMS server mainline module, which is generated when you run the JCL in *orbixhlq*.DEMOS.IMS.PLI.BLD.JCL(SIMPLIDL). (The IMS server mainline code is not shipped with the product. You must run the SIMPLIDL JCL to generate it.)
orbixhlq.DEMOS.IMS.PLI.SRC (SIMPLES)	This is the source code for the IMS server implementation module.
orbixhlq.DEMOS.IMS.PLI.SRC (SIMPLECL)	This is the source code for the IMS client module.
orbixhlq.DEMOS.IMS.PLI.BLD.JCL (SIMPLIDL)	This JCL runs the Orbix IDL compiler. See "Orbix IDL Compiler" on page 59 for more details of this JCL and how to use it.
orbixhlq.DEMOS.IMS.PLI.BLD.JCL (SIMPLESB)	This JCL compiles and links the IMS server mainline and IMS server implementation modules to create the SIMPLE server program.
orbixhlq.DEMOS.IMS.PLI.BLD.JCL (SIMPLECB)	This JCL compile the IMS client module to create the SIMPLE client program.
orbixhlq.DEMOS.IMS.PLI.BLD.JCL (SIMPLREG)	This JCL registers the IDL in the Interface Repository (IFR).
orbixhlq.DEMOS.IMS.PLI.BLD.JCL (SIMPLIOR)	This JCL obtains the IMS server's IOR (from the IMS server adapter). A client of the IMS server requires the IMS server's IOR, to locate the server object.

Table 5: *Supplied Code and JCL (Sheet 2 of 2)*

Location	Description
orbixhlq.DEMOS.IMS.PLI.BLD.JCL (UPDTCONF)	This JCL adds the following configuration entry to the configuration member: `initial_references:SimpleObject:reference="IOR…";` This configuration entry specifies the IOR that the IMS client uses to contact the batch server. The IOR that is set as the value for this configuration entry is the IOR that is published in *orbixhlq*.DEMOS.IORS(SIMPLE) when you run the batch server. The object reference for the server is represented to the demonstration IMS client as a corbaloc URL string in the form `corbaloc:rir:/SimpleObject`. This form of corbaloc URL string requires the use of the `initial_references:SimpleObject:reference="IOR…"` configuration entry. Other forms of corbaloc URL string can also be used (for example, the IIOP version, as demonstrated in the nested sequences demonstration supplied with your product installation). See “STR2OBJ” on page 404 for more details of the various forms of corbaloc URL strings and the ways you can use them.
orbixhlq.JCL(MFCLA)	This JCL configures and runs the client adapter.
orbixhlq.JCL(IMSA)	This JCL configures and runs the IMS server adapter.

Supplied include members

Table 6 provides a summary in alphabetic order of the various include members supplied with your product installation that are relevant to IMS application development. Again, *orbixhlq* represents your installation’s high-level qualifier.

Table 6: *Supplied Include Members (Sheet 1 of 3)*

Location	Description
orbixhlq.INCLUDE.PLINCL(CHKCLIMS)	This is relevant to IMS clients only. It contains a PL/I function that can be called by the client, to check if a system exception has occurred, and to report that system exception.

Table 6: *Supplied Include Members (Sheet 2 of 3)*

Location	Description
orbixhlq.INCLUDE.PLINCL(CHKERRS)	This is relevant to IMS servers. It contains a PL/I function that can be called by the IMS server, to check if a system exception has occurred, and to report that system exception.
orbixhlq.INCLUDE.PLINCL(CORBA)	This is relevant to both IMS clients and servers. It contains common PL/I runtime variables. It includes the CORBACOM include member by default. It also includes the CORBASV include member, if the client module contains the line %client_only='yes';.
orbixhlq.INCLUDE.PLINCL(CORBACOM)	This is relevant to both IMS clients and servers. It contains common PL/I runtime function definitions that can be used both by clients and servers.
orbixhlq.INCLUDE.PLINCL(CORBASV)	This is relevant to IMS servers. It contains PL/I runtime function definitions that can be used by servers.
orbixhlq.INCLUDE.PLINCL(DISPINIT)	This is relevant to IMS servers only. It retrieves the current request information into the REQINFO structure via PODREQ. From REQINFO the operation to be performed by the server is retrieved via a call to STRGET.
orbixhlq.INCLUDE.PLINCL(DLIDATA)	This is relevant to IMS clients only. It contains structures to facilitate reading from and writing to the IMS message queue via iopcb_ptr. It contains a a PL/I function called write_dc_text that facilitates writing messages to the IMS output message queue. It does this by using the supplied IBM routine (interface) PLITDLI to make an IMS DC (data communications) call that specifies the common IMS function command ISRT (insert). The DLIDATA member contains all the declarations needed for the supplied PL/I client demonstration in IMS.
orbixhlq.INCLUDE.PLINCL(EXCNAME)	This is relevant to both IMS clients and servers. It contains a PL/I function called CORBA_EXC_NAME that returns the system exception name for the system exception being raised (that is, it maps Orbix exceptions to human-readable strings). EXCNAME is used by CHKERRS and CHKCLIMS.

Table 6: *Supplied Include Members (Sheet 3 of 3)*

Location	Description
orbixhlq.INCLUDE.PLINCL(GETUNIQ)	This is relevant to IMS clients only. It contains a PL/I function that can be called by the client, to retrieve specific IMS segments. It does this by using the supplied IBM routine (interface) `PLITDLI` to make an IMS DC (data communications) call that specifies the `GU` (get unique) function command.
orbixhlq.INCLUDE.PLINCL(IMSPCB)	This is relevant to IMS servers only. It is used in IMS server modules. It contains three structures: `pcblist`, `io_pcb`, and `alt_pcb`. The `pcblist` structure is static, and it allows access to the PCB pointers from anywhere within the PL/I IMS server code. The `io_pcb` and `alt_pcb` structures are based onto `pcblist.io_pcb_ptr` and `pcblist.alt_pcb_ptr` respectively. **Note:** The supplied demonstration omits the line `%include IMSPCB`, which means it does not make use of the variables declared in this include member.
orbixhlq.INCLUDE.PLINCL(URLSTR)	This is relevant to clients only. It contains a PL/I representation of the corbaloc URL IIOP string format. A client can call `STR2OBJ` to convert the URL into an object reference. See "STR2OBJ" on page 404 for more details.
orbixhlq.DEMOS.IMS.PLI.PLINCL	This PDS is used to store all IMS include members that are generated when you run the JCL to run the Orbix IDL compiler for the supplied demonstrations. It also contains helper procedures for the nested sequences demonstration.
orbixhlq.DEMOS.IMS.MFAMAP	This PDS is relevant to IMS servers only. It is empty at installation time. It is used to store the IMS server adapter mapping member generated when you run the JCL to run the Orbix IDL compiler for the supplied demonstrations. The contents of the mapping member are the fully qualifed interface name followed by the operation name followed by the IMS transaction name (for example, `(Simple/SimpleObject,call_me,SIMPLESV)`. See the *IMS Adapters Administrator's Guide* for more details about generating server adapter mapping members.

Checking JCL components

When creating either the IMS client or server `SIMPLE` application, check that each step involved within the separate JCL components completes with a condition code not greater than `4`. If the condition codes are greater than `4`, establish the point and cause of failure. The most likely cause is the site-specific JCL changes required for the compilers. Ensure that each high-level qualifier throughout the JCL reflects your installation.

Developing the Application Interfaces

Overview

This section describes the steps you must follow to develop the IDL interfaces for your application. It first describes how to define the IDL interfaces for the objects in your system. It then describes how to run the IDL compiler. Finally it provides an overview of the PL/I include members, server source code, and IMS server adapter mapping member that you can generate via the IDL compiler.

Steps to develop application interfaces

The steps to develop the interfaces to your application are:

Step	Action
1	Define public IDL interfaces to the objects required in your system. See "Defining IDL Interfaces" on page 57.
2	Run the Orbix IDL compiler to generate PL/I include members, server source, and server mapping member. See "Orbix IDL Compiler" on page 59.

Defining IDL Interfaces

Defining the IDL

The first step in writing any Orbix program is to define the IDL interfaces for the objects required in your system. The following is an example of the IDL for the `SimpleObject` interface that is supplied in *orbixhlq*.DEMOS.IDL(SIMPLE):

```
// IDL
module Simple
{
    interface SimpleObject
    {
        void
        call_me();
    };
};
```

Explanation of the IDL

The preceding IDL declares a `SimpleObject` interface that is scoped (that is, contained) within the `Simple` module. This interface exposes a single `call_me()` operation. This IDL definition provides a language-neutral interface to the CORBA `Simple::SimpleObject` type.

How the demonstration uses this IDL

For the purposes of the demonstrations in this chapter, the `SimpleObject` CORBA object is implemented in PL/I in the supplied simple server application. The server application creates a persistent server object of the `SimpleObject` type, and publishes its object reference to a PDS member. The client invokes the `call_me()` operation on the `SimpleObject` object, and then exits.

The batch demonstration client of the IMS demonstration server locates the `SimpleObject` object by reading the interoperable object reference (IOR) for the IMS server adapter from *orbixhlq*.DEMOS.IORS(SIMPLE). In this case, the IMS server adapter IOR is published to *orbixhlq*.DEMOS.IORS(SIMPLE) when you run *orbixhlq*.DEMOS.IMS.PLI.BLD.JCL(SIMPLIOR).

The IMS demonstration client of the batch demonstration server locates the `SimpleObject` object by reading the IOR for the batch server from *orbixhlq*.DEMOS.IORS(SIMPLE). In this case, the batch server IOR is published to *`orbixhlq`*`.DEMOS.IORS(SIMPLE)` when you run the batch server. The object reference for the server is represented to the demonstration IMS client as a corbaloc URL string in the form `corbaloc:rir:/SimpleObject`.

Orbix IDL Compiler

The Orbix IDL compiler

This subsection describes how to use the Orbix IDL compiler to generate PL/I include members, server source, and the IMS server adapter mapping member from IDL.

Note: Generation of PL/I include members is relevant to both IMS client and server development. Generation of server source and the IMS server adapter mapping member is relevant only to IMS server development.

Orbix IDL compiler configuration

The Orbix IDL compiler uses the Orbix configuration member for its settings. The `SIMPLIDL` JCL that runs the compiler uses the configuration member *`orbixhlq`*`.CONFIG(IDL)`. See “Orbix IDL Compiler” on page 233 for more details.

Example of the SIMPLIDL JCL

The following is the supplied JCL to run the Orbix IDL compiler for the IMS `SIMPLE` demonstration:

```
//SIMPLIDL JOB   (),
//         CLASS=A,
//         MSGCLASS=X,
//         MSGLEVEL=(1,1),
//         REGION=0M,
//         TIME=1440,
//         NOTIFY=&SYSUID,
//         COND=(4,LT)
//*-------------------------------------------------------------
//* Orbix - Generate the PL/I IMS server files for Simple Demo
//*-------------------------------------------------------------
//         JCLLIB ORDER=(orbixhlq.PROCS)
//         INCLUDE MEMBER=(ORXVARS)
//*
//* Make the following changes before running this JCL:
//*
//* 1.   Change 'SET DOMAIN='DEFAULT@' to your configuration
//*      domain name.
//*
//            SET DOMAIN='DEFAULT@'
//*
```

```
//IDLPLI   EXEC ORXIDL,
//         SOURCE=SIMPLE,
//         IDL=&ORBIX..DEMOS.IDL,
//         IDLPARM='-pli:-TIMS -mfa:-tSIMPLESV'
//*        IDLPARM='-pli:-V'
//IDLMFA   DD DISP=SHR, DSN=&ORBIX..DEMOS.IMS.MFAMAP
//ITDOMAIN DD DSN=&ORBIX..CONFIG(&DOMAIN),DISP=SHR
```

Explanation of the SIMPLIDL JCL

In the preceding JCL example, the lines `IDLPARM='-pli:-V'` and `IDLPARM='-pli:-TIMS -mfa:-tSIMPLESV'` are mutually exclusive. The line `IDLPARM='-pli:-TIMS -mfa:-tSIMPLESV'` is relevant to IMS server development and generates:

- PL/I include members via the `-pli` argument.
- IMS server mainline code via the `-TIMS` argument.
- IMS server adapter mapping member via the `-mfa:-t`*`tran_name`* arguments.

Note: Because IMS server implementation code is already supplied for you, the `-S` argument is not specified by default.

The line `IDLPARM='-pli:-V'` in the preceding JCL is relevant to IMS client development and generates only PL/I include members, because it only specifies the `-pli:-V` arguments (The `-V` argument prevents generation of PL/I server mainline source code.)

Note: The Orbix IDL compiler does not generate PL/I client source code.

Specifying what you want to generate

To indicate which of these lines you want `SIMPLIDL` to recognize, comment out the line you do not want to use, by placing an asterisk at the start of that line. By default, as shown in the preceding example, the JCL is set to generate PL/I include members, server mainline code, and an IMS server adapter mapping member. Alternatively, if you choose to comment out the line that has the `-pli:-TIMS -mfa:-tSIMPLESV` arguments, the IDL compiler only generates PL/I include members.

See "Orbix IDL Compiler" on page 233 for more details of the Orbix IDL compiler and the JCL used to run it.

Running the Orbix IDL compiler

After you have edited the `SIMPLIDL` JCL according to your requirements, you can run the Orbix IDL compiler by submitting the following job:

```
orbixhlq.DEMOS.IMS.PLI.BLD.JCL(SIMPLIDL)
```

Generated PL/I Include Members, Source, and Mapping Member

Overview

This subsection describes all the PL/I include members, server source, and IMS server adapter mapping member that the Orbix IDL compiler can generate from IDL definitions.

Note: The generated PL/I include members are relevant to both IMS client and server development. The generated source and adapter mapping member are relevant only to IMS server development. The IDL compiler does not generate PL/I client source.

Member name restrictions

Generated PL/I source code, include, and mapping member names are all based on the IDL member name. If the IDL member name exceeds six characters, the Orbix IDL compiler uses only the first six characters of the IDL member name when generating the other member names. This allows space for appending a one-character suffix to each generated member name, while allowing it to adhere to the seven-character maximum size limit for PL/I external procedure names, which are based by default on the generated member names.

How IDL maps to PL/I include members

Each IDL interface maps to a group of PL/I structures. There is one structure defined for each IDL operation. A structure contains each of the parameters for the relevant IDL operation in their corresponding PL/I representation. See "IDL-to-PL/I Mapping" on page 177 for details of how IDL types map to PL/I.

Attributes map to two operations (`get` and `set`), and readonly attributes map to a single `get` operation.

Generated PL/I include members

Table 7 shows the PL/I include members that the Orbix IDL compiler generates, based on the defined IDL.

Table 7: *Generated PL/I Include Members (Sheet 1 of 2)*

Copybook	JCL Keyword Parameter	Description
`*idlmembername*D`	`COPYLIB`	This include member contains a select statement that determines which server implementation procedure is to be called, based on the interface name and operation received.
`*idlmembername*L`	`COPYLIB`	This include member contains structures and procedures used by the PL/I runtime to read and store data into the operation parameters. This member is automatically included in the `*idlmembername*X` include member.
`*idlmembername*M`	`COPYLIB`	This include member contains declarations and structures that are used for working with operation parameters and return values for each interface defined in the IDL member. The structures use the based PL/I structures declared in the `*idlmembername*T` include member. This member is automatically included in the `*idlmembername*I` include member.

Table 7: *Generated PL/I Include Members (Sheet 2 of 2)*

Copybook	JCL Keyword Parameter	Description
*idlmembername*T	COPYLIB	This include member contains the based structure declarations that are used in the *idlmembername*M include member. This member is automatically included in the *idlmembername*M include member.
*idlmembername*X	COPYLIB	This include member contains structures that are used by the PL/I runtime to support the interfaces defined in the IDL member. This member is automatically included in the *idlmembername*V source code member.
*idlmembername*D	COPYLIB	This include member contains a select statement for calling the correct procedure for the requested operation. This include member is automatically included in the *idlmembername*I source code member.

Generated server source members

Table 8 shows the server source code members that the Orbix IDL compiler generates, based on the defined IDL.

Table 8: *Generated Server Source Code Members*

Member	JCL Keyword Parameter	Description
`*idlmembername*I`	`IMPL`	This is the IMS server implementation source code member. It contains procedure definitions for all the callable operations. This is only generated if you specify both the `-S` and `-TIMS` arguments with the IDL compiler.
`*idlmembername*V`	`IMPL`	This is the IMS server mainline source code member. It is generated by default. However, you can use the -V argument with the IDL compiler, to prevent generation of this member.

Note: For the purposes of this example, the `SIMPLEI` server implementation member is already provided in your product installation. Therefore, the `-S` IDL compiler argument used to generate it is not specified in the supplied `SIMPLIDL` JCL. The `SIMPLEV` server mainline is not already provided, so the `-V` argument, which prevents generation of server mainline code, is not specified in the supplied JCL. See "Orbix IDL Compiler" on page 233 for more details of the IDL compiler arguments used to generate, and prevent generation of, IMS server source code.

Generated server adapter mapping member

Table 9 shows the IMS server adapter mapping member that the Orbix IDL compiler generates, based on the defined IDL.

Table 9: *Generated IMS Server Adapter Mapping Member*

Copybook	JCL Keyword Parameter	Description
*idlmembername*A	MEMBER	This is a simple text file that determines what interfaces and operations the IMS server adapter supports, and the IMS transaction names to which the IMS server adapter should map each IDL operation.

Location of demonstration copybooks and mapping member

You can find examples of the include members, server source, and IMS server adapter mapping member generated for the SIMPLE demonstration in the following locations:

- *orbixhlq*.DEMOS.IMS.PLI.PLINCL(SIMPLED)
- *orbixhlq*.DEMOS.IMS.PLI.PLINCL(SIMPLEL)
- *orbixhlq*.DEMOS.IMS.PLI.PLINCL(SIMPLEM)
- *orbixhlq*.DEMOS.IMS.PLI.PLINCL(SIMPLET)
- *orbixhlq*.DEMOS.IMS.PLI.PLINCL(SIMPLEX)
- *orbixhlq*.DEMOS.IMS.PLI.SRC(SIMPLEV)
- *orbixhlq*.DEMOS.IMS.PLI.SRC(SIMPLEI)
- *orbixhlq*.DEMOS.IMS.MFAMAP(SIMPLEA)

Note: Except for the SIMPLEI member, none of the preceding elements are shipped with your product installation. They are generated when you run *orbixhlq*.DEMOS.IMS.PLI.BLD.JCL(SIMPLIDL), to run the Orbix IDL compiler.

Developing the IMS Server

Overview

This section describes the steps you must follow to develop the IMS server executable for your application. The IMS server developed in this example will be contacted by the simple batch client demonstration.

Steps to develop the server

The steps to develop the server application are:

Step	Action
1	“Writing the Server Implementation” on page 68.
2	“Writing the Server Mainline” on page 71.
3	“Building the Server” on page 74.
4	“Preparing the Server to Run in IMS” on page 75.

Writing the Server Implementation

The server implementation module

You must implement the server interface by writing a PL/I implementation module that implements each operation defined to the operation section in the *idlmembername*T include member. For the purposes of this example, you must write a PL/I procedure that implements each operation in the `SIMPLET` include member. When you specify the `-S` and `-TIMS` arguments with the Orbix IDL compiler, it generates a skeleton server implementation module, in this case called `SIMPLEI`, which is a useful starting point.

> **Note:** For the purposes of this demonstration, the IMS server implementation module, `SIMPLEI`, is already provided for you, so the `-S` argument is not specified in the JCL that runs the IDL compiler.

Example of the IMS SIMPLEI module

The following is an example of the IMS `SIMPLEI` module (with the header comment block omitted for the sake of brevity):

Table 10: *The SIMPLEI Demonstration Module (Sheet 1 of 2)*

```
SIMPLEI: PROC;

/*The following line enables the runtime to call this procedure*/
1 DISPTCH: ENTRY;

dcl (addr,low,sysnull)                   builtin;

%include CORBA;
%include CHKERRS;
%include DLIDATA;
%include SIMPLEM;
2 %include DISPINIT;

/* ================ Start of global user code =============== */
/* ================= End of global user code ================ */

/* ----------------------------------------------------------*/
/*                                                           */
/* Dispatcher : select(operation)                            */
/*                                                           */
/*-----------------------------------------------------------*/
3 %include SIMPLED;
```

Table 10: *The SIMPLEI Demonstration Module (Sheet 2 of 2)*

```
/*-----------------------------------------------------------*/
/* Interface:                                                */
/*     Simple/SimpleObject                                   */
/*                                                           */
/* Mapped name:                                              */
/*     Simple_SimpleObject                                   */
/*                                                           */
/* Inherits interfaces:                                      */
/*     (none)                                                */
/*-----------------------------------------------------------*/
/*-----------------------------------------------------------*/
/* Operation:       call_me                                  */
/* Mapped name:     call_me                                  */
/* Arguments:       None                                     */
/* Returns:         void                                     */
/*-----------------------------------------------------------*/
4 proc_Simple_SimpleObject_c_c904: PROC(p_args);

dcl p_args                     ptr;
5 dcl 1_args                     aligned based(p_args)
                             like Simple_SimpleObject_c_ba77_type;

/* ============ Start of operation specific code ============= */
6 put skip list('Operation call_me() called');
put skip;
/* ============== End of operation specific code ============= */

END proc_Simple_SimpleObject_c_c904;

END SIMPLEI;
```

Explanation of the IMS SIMPLEI module

The IMS `SIMPLEI` module can be explained as follows:

1. When an incoming request arrives from the network, it is processed by the ORB and a call is made from the PL/I runtime to the `DISPTCH` entry point.
2. Within the `DISPINIT` include member, `PODREQ` is called to provide information about the current invocation request, which is held in the `REQINFO` structure. `PODREQ` is called once for each operation invocation

after a request has been dispatched to the server. STRGET is then called to copy the characters in the unbounded string pointer for the operation name into the PL/I string that represents the operation name.

3. The SIMPLED include member contains a select statement that determines which procedure within SIMPLEI is to be called, given the operation name and interface name passed to SIMPLEI. It calls PODGET before the call to the server procedure, which fills the appropriate PL/I structure declared in the main include member, SIMPLEM, with the operation's incoming arguments. It then calls PODPUT after the call to the server procedure, to send out the operation's outgoing arguments.
4. The procedural code containing the server implementation for the call_me operation.
5. Each operation has an argument structure and these are declared in the typecode include member, SIMPLET. If an operation does not have any parameters or return type, such as call_me, the structure only contains a structure with a dummy char.
6. This is a sample of the server implementation code for call_me. It is the only part of the SIMPLEI member that is not automatically generated by the Orbix IDL compiler.

Note: An operation implementation should not call PODGET or PODPUT. These calls are made within the SIMPLED include member generated by the Orbix IDL compiler.

Location of the IMS SIMPLEI module

You can find a complete version of the IMS SIMPLEI server implementation module in *orbixhlq*.DEMOS.IMS.PLI.SRC(SIMPLEI).

Writing the Server Mainline

The server mainline module

The next step is to write the server mainline module in which to run the server implementation. For the purposes of this example, when you specify the `-TIMS` argument with the Orbix IDL compiler, it generates a module called `SIMPLEV`, which contains the server mainline code.

Note: Unlike the batch server mainline, the IMS server mainline does not have to create and store stringified object references (IORs) for the interfaces that it implements, because this is handled by the IMS server adapter.

Example of the IMS SIMPLEV module

The following is an example of the IMS `SIMPLEV` module:

Table 11: *The SIMPLEV Demonstration Module (Sheet 1 of 2)*

```
SIMPLEV: PROC(IO_PCB_PTR,ALT_PCB_PTR) OPTIONS(MAIN NOEXECOPS);
dcl (io_pcb_ptr,alt_pcb_ptr)   ptr;

dcl arg_list                   char(01)        init('');
dcl arg_list_len               fixed bin(31)   init(0);
dcl orb_name                  char(10)       init('simple_orb');
dcl orb_name_len               fixed bin(31)   init(10);
dcl srv_name                   char(256) var;
dcl server_name                char(07)        init('simple ');
dcl server_name_len            fixed bin(31)   init(6);

dcl Simple_SimpleObject_objid  char(27)
    init('Simple/SimpleObject_object');
dcl Simple_SimpleObject_obj    ptr;
dcl SYSPRINT                   file stream output;
dcl (addr,length,low,sysnull)  builtin;

%include CORBA;
%include CHKERRS;
%include IMSPCB;
%include SIMPLET;
%include SIMPLEX;

pcblist.io_pcb_ptr  =  io_pcb_ptr;
```

Table 11: *The SIMPLEV Demonstration Module (Sheet 2 of 2)*

```
  pcblist.alt_pcb_ptr  =  alt_pcb_ptr;

  pcblist.num_db_pcbs  =  0;

  alloc pod_status_information set(pod_status_ptr);

1 call podstat(pod_status_ptr);
  if check_errors('podstat') ^= completion_status_yes then return;

  /* Initialize the server connection to the ORB                 */
2 call orbargs(arg_list,arg_list_len,orb_name,orb_name_len);
  if check_errors('orbargs') ^= completion_status_yes then return;

3 call podsrvr(server_name, server_name_len);
  if check_errors('podsrvr') ^= completion_status_yes then return;

  /* Register interface : Simple/SimpleObject                    */
4 call podreg(addr(Simple_SimpleObject_interface));
  if check_errors('podreg';) ^= completion_status_yes then return;

5 call objnew(server_name,
              Simple_SimpleObject_intf,
              Simple_SimpleObject_objid,
              Simple_SimpleObject_obj);
  if check_errors('objnew') ^= completion_status_yes then return;

  /* Server is now ready to accept requests                      */
6 call podrun;
  if check_errors('podrun') ^= completion_status_yes then return;

7 call objrel(Simple_SimpleObject_obj);
  if check_errors('objrel') ^= completion_status_yes then return;

  free pod_status_information;

  END SIMPLEV;
```

Explanation of the IMS SIMPLEV module

The IMS `SIMPLEV` module can be explained as follows:

1. `PODSTAT` is called to register the `POD_STATUS_INFORMATION` block that is contained in the `CORBA` include member. Registering the `POD_STATUS_INFORMATION` block allows the PL/I runtime to populate it

with exception information, if necessary. If `completion_status` is set to zero after a call to the PL/I runtime, this means that the call has completed successfully.

2. `ORBARGS` is called to initialize a connection to the ORB.
3. `PODSRVR` is called to set the server name.
4. `PODREG` is called to register the IDL interface, `SimpleObject`, with the PL/I runtime.
5. `OBJNEW` is called to create a persistent server object of the `SimpleObject` type, with an object ID of `my_simple_object`.
6. `PODRUN` is called, to enter the `ORB::run()` loop, to allow the ORB to receive and process client requests. This then processes the CORBA request that the IMS server adapter sends to IMS. If the transaction has been defined as WFI, multiple requests can be processed in the `PODRUN` loop; otherwise, `PODRUN` processes only one request.
7. `OBJREL` is called to ensure that the servant object is released properly.

See the preface of this guide for details about the compilers that this product supports.

Location of the IMS SIMPLESV module

You can find a complete version of the IMS `SIMPLEV` server mainline module in *`orbixhlq`*`.DEMOS.IMS.PLI.SRC(SIMPLEV)` after you have run *`orbixhlq`*`.DEMOS.IMS.PLI.BLD.JCL(SIMPLIDL)` to run the Orbix IDL compiler.

Building the Server

Location of the JCL

Sample JCL used to compile and link the IMS server mainline and server implementation is in *orbixhlq*.DEMOS.IMS.PLI.BLD.JCL(SIMPLESB).

Resulting load module

When this JCL has successfully executed, it results in a load module that is contained in *orbixhlq*.DEMOS.IMS.PLI.LOAD(SIMPLESV).

Preparing the Server to Run in IMS

Overview

This section describes the required steps to allow the server to run in an IMS region. These steps assume you want to run the IMS server against a batch client. When all the steps in this section have been completed, the server is started automatically within IMS, as required.

Steps

The steps to enable the server to run in an IMS region are:

Step	Action
1	Define a transaction definition for IMS.
2	Provide the IMS server load module to an IMS region.
3	Generate mapping member entries for the IMS server adapter.
4	Add the IDL to the Interface Repository. **Note:** For the purposes of this demonstration, the IFR is used as the source of type information.
5	Obtain the IOR for use by the client program.

Step 1—Defining transaction definition for IMS

A transaction definition must be created for the server, to allow it to run in IMS. The following is the transaction definition for the supplied demonstration:

```
    APPLCTN    GPSB=SIMPLESV,                                      x
          PGMTYPE=(TP,,2),                                          x
          SCHDTYP=PARALLEL
    TRANSACT   CODE=SIMPLESV,                                      x
          EDIT=(ULC)
```

Step 2—Providing load module to IMS region

Ensure that the *orbixhlq*.DEMOS.IMS.PLI.LOAD PDS is added to the STEPLIB for the IMS region that is to run the transaction, or copy the SIMPLESV load module to a PDS in the STEPLIB of the relevant IMS region.

Step 3—Generating mapping member entries

The IMS server adapter requires mapping member entries, so that it knows which IMS transaction should be run for a particular interface and operation. The mapping member entry for the supplied example is contained in *orbixhlq*.DEMOS.IMS.MFAMAP(SIMPLEA) (after you run the IDL compiler) and appears as follows:

```
(Simple/SimpleObject,call_me,SIMPLESV)
```

The generation of a mapping member for the IMS server adapter is performed by the *orbixhlq*.DEMOS.IMS.PLI.BLD.JCL(SIMPLIDL) JCL. The -mfa:-t*tran_name* argument with the IDL compiler generates the mapping member. For the purposes of this example, *tran_name* is replaced with SIMPLESV. An IDLMFA DD statement must also be provided in the JCL, to specify the PDS into which the mapping member is generated. See the *IMS Adapters Administrator's Guide* for full details about IMS server adapter mapping members.

Step 4—Adding IDL to Interface Repository

The IMS server adapter needs to be able to obtain operation signatures for the PL/I server. For the purposes of this demonstration, the IFR is used to retrieve this type information. This type information is necessary so that the adapter knows what data types it has to marshal into IMS for the server, and what data types it can expect back from the IMS transaction. Ensure that the relevant IDL for the server has been added to (that is, registered with) the Interface Repository before the IMS server adapter is started.

To add IDL to the Interface Repository, the Interface Repository must be running. You can use the JCL in *orbixhlq*.JCL(IFR) to start it. The Interface Repository uses the configuration settings in the Orbix configuration member, *orbixhlq*.CONFIG(DEFAULT@).

The following JCL that adds IDL to the Interface Repository is supplied in *orbixhlq*.DEMOS.IMS.PLI.BLD.JCL(SIMPLEREG):

```
//          JCLLIB ORDER=(orbixhlq.PROCS)
//          INCLUDE MEMBER=(ORXVARS)
//*
//* Make the following changes before running this JCL:
//*
//* 1.   Change 'SET DOMAIN='DEFAULT@' to your configuration
//*      domain name.
//*
//            SET DOMAIN='DEFAULT@'
//*
//IDLPLI   EXEC ORXIDL,
//          SOURCE=SIMPLE,
//          IDL=&ORBIX..DEMOS.IDL,
//          IDLPARM='-R'
//ITDOMAIN DD DSN=&ORBIX..CONFIG(&DOMAIN),DISP=SHR
```

Note: An alternative to using the IFR is to use type information files. These are an alternative method of providing IDL interface information to the IMS server adapter. Type information files can be generated as part of the `-mfa` plug-in to the IDL compiler. See the *IMS Adapters Administrator's Guide* for more details about how to generate them. The use of type information files would render this step unnecessary; however, the use of the IFR is recommended for the purposes of this demonstration.

Step 5—Obtaining the server adapter IOR

The final step is to obtain the IOR that the batch client needs to locate the IMS server adapter. Before you do this, ensure all of the following:

- The Interface Repository is running and contains the relevant IDL. See "Step 4—Adding IDL to Interface Repository" on page 76 for details of how to start it, if it is not already running.
- The IMS server adapter is running. See the *IMS Adapters Administrator's Guide* for more details about how to start the IMS server adapter.
- The IMS server adapter mapping member contains the relevant mapping entries. For the purposes of this example, ensure that the *orbixhlq*.DEMOS.IMS.MFAMAP(SIMPLEA) mapping member is being used. See the *IMS Adapters Administrator's Guide* for details about IMS server adapter mapping members.

Now submit *orbixhlq*.DEMOS.IMS.PLI.BLD.JCL(SIMPLIOR), to obtain the IOR that the batch client needs to locate the IMS server adapter. This JCL includes the resolve command, to obtain the IOR. The following is an example of the SIMPLIOR JCL:

```
//         JCLLIB ORDER=(orbixhlq.PROCS)
//         INCLUDE MEMBER=(ORXVARS)
//*
//* Request the IOR for the IMS 'simple_persistent' server
//* and store it in a PDS for use by the client.
//*
//* Make the following changes before running this JCL:
//*
//* 1.  Change 'SET DOMAIN='DEFAULT@' to your configuration
//*     domain name.
//*
//           SET DOMAIN='DEFAULT@'
//*
//REG      EXEC PROC=ORXADMIN,
//  PPARM='mfa resolve Simple/SimpleObject > DD:IOR'
//IOR DD DSN=&ORBIX..DEMOS.IORS(SIMPLE),DISP=SHR
//ORBARGS DD *
-ORBname iona_utilities.imsa
/*
//ITDOMAIN DD DSN=&ORBIX..CONFIG(&DOMAIN),DISP=SHR
```

When you submit the SIMPLIOR JCL, it writes the IOR for the IMS server adapter to *orbixhlq*.DEMOS.IORS(SIMPLE).

Developing the IMS Client

Overview

This section describes the steps you must follow to develop the IMS client executable for your application. The IMS client developed in this example will connect to the simple batch server demonstration.

Note: The Orbix IDL compiler does not generate PL/I client stub code.

Steps to develop the client

The steps to develop and run the client application are:

Step	Action
1	"Writing the Client" on page 80.
2	"Building the Client" on page 84.
3	"Preparing the Client to Run in IMS" on page 85.

Writing the Client

The client program

The next step is to write the client program, to implement the IMS client. This example uses the supplied `SIMPLECL` client demonstration.

Example of the SIMPLEC module

The following is an example of the IMS `SIMPLEC` module:

Table 12: *The SIMPLEC Demonstration Module (Sheet 1 of 3)*

```
SIMPLEC: PROC (IO_PCB_PTR, ALT_PCB_PTR) OPTIONS(MAIN NOEXECOPS);

 dcl (io_pcb_ptr, alt_pcb_ptr)     ptr;

 %client_only='yes';

 dcl (addr index,low,substr,sysnull,length)   builtin;

 dcl arg_list                         char(40)       init('');
 dcl arg_list_len                     fixed bin(31)  init(38);
 dcl orb_name                         char(10)
                                      init('simple_orb');
 dcl orb_name_len                     fixed bin(31)  init(10);

 dcl sysprint                         file stream output;

1 dcl simple_url              char(27)
                     init('corbaloc:rir:/SimpleObject ');
 dcl simple_url_ptr                   ptr  init(sysnull());
 dcl Simple_SimpleObject_obj          ptr;

 %include CORBA;
 %include IMSPCB;
2 %include DLIDATA;
 %include GETUNIQ;
3 %include CHKCLIMS;
 %include SIMPLEM;
 %include SIMPLEX;

 pcblist.io_pcb_ptr  =  io_pcb_ptr;
 pcblist.alt_pcb_ptr =  alt_pcb_ptr;
 call get_uniq;
 /* Initialize the PL/I runtime status information block */
 alloc pod_status_information set(pod_status_ptr);
```

Table 12: *The SIMPLEC Demonstration Module (Sheet 2 of 3)*

```
4  call podstat(pod_status_ptr);

   /* Initialize our ORB */
5  call orbargs(arg_list,
                arg_list_len,
                orb_name,
                orb_name_len);
   if check_errors('orbargs') ^= completion_status_yes then
      return;

   /* Register the SimpleObject interface with the PL/I runtime */
6  call podreg(addr(Simple_SimpleObject_interface));
   if check_errors('podreg') ^= completion_status_yes then
      return;

   /* Create an object reference from the server's IOR */
   /* so we can make calls to the server               */
7  call strset(simple_url_ptr,
               simple_url,
               length(simple_url));
   if check_errors('strset') ^= completion_status_yes then
      return;

8  call str2obj(simple_url_ptr,Simple_SimpleObject_obj);
   if check_errors('str2obj') ^= completion_status_yes then
      return;

   /* Now we are ready to start making server requests */

   put skip list('simple_persistent demo');
   put skip list('======================');

   /* Call operation call_me */
   /* As this is a very simple function, there aren't any    */
   /* parameters. So instead we pass in the generated dummy  */
   /* structure created for this operation.                  */
   put skip list('Calling operation call_me...');
9  call podexec(Simple_SimpleObject_obj,
                Simple_SimpleObject_call_me,
                addr(Simple_SimpleObject_c_ba77_args),
                no_user_exceptions);
   if check_errors('podexec') ^= completion_status_yes then
     return;
```

Table 12: *The SIMPLEC Demonstration Module (Sheet 3 of 3)*

```
   put skip list('Operation call_me completed (no results to
     display)');
   put skip;
   put skip list('End of the simple_persistent demo');
   put skip;
   dc_text = 'Simple Transaction completed';
   call write_dc_text(dc_text,38);

   /* Free the simple_persistent object reference */
10 call objrel(Simple_SimpleObject_obj);
   if check_errors('objrel') ^= completion_status_yes then
      return;

   free pod_status_information;

   END SIMPLEC;
```

Explanation of the SIMPLEC module

The IMS `SIMPLEC` module can be explained as follows:

1. `simple_url` defines a corbaloc URL string in the `corbaloc:rir` format. This string identifies the server with which the client is to communicate. This string can be passed as a parameter to `STR2OBJ`, to allow the client to retrieve an object reference to the server. See point **8** about `STR2OBJ` for more details.
2. The `write_dc_text` function is provided in the `DLIDATA` include member. This function allows messages generated by the demonstrations to be written to the IMS message queue.
3. A special error-checking include member is used for IMS clients.
4. `PODSTAT` is called to register the `POD_STATUS_INFORMATION` block that is contained in the `CORBA` include member. Registering the `POD_STATUS_INFORMATION` block allows the PL/I runtime to populate it with exception information, if necessary. If `completion_status` is set to zero after a call to the PL/I runtime, this means that the call has completed successfully.

   The `check_errors` function can be used to test the status of any Orbix call. It tests the value of the `exception_number` in `pod_status_information`. If its value is zero, it means the call was successful. Otherwise, `check_errors` prints out the system exception number and message, and the program ends at that point. The `check_errors` function should be called after every PL/I runtime call to ensure the call completed successfully.

5. `ORBARGS` is called to initialize a connection to the ORB.
6. `PODREG` is called to register the IDL interface with the Orbix PL/I runtime.
7. `STRSET` is called to create an unbounded string to which the stringified object reference is copied.
8. `STR2OBJ` is called to create an object reference to the server object. This must be done to allow operation invocations on the server. In this case, the client identifies the target object, using a corbaloc URL string in the form `corbaloc:rir:/SimpleObject` (as defined in point **1**). See "STR2OBJ" on page 404 for more details of the various forms of corbaloc URL strings and the ways you can use them.
9. After the object reference is created, `PODEXEC` is called to invoke operations on the server object represented by that object reference. You must pass the object reference, the operation name, the argument description packet, and the user exception buffer. If the call does not have a user exception defined (as in the preceding example), the `no_user_exceptions` variable is passed in instead. The operation name must be terminated with a space. The same argument description is used by the server. For ease of use, string identifiers for operations are defined in the `SIMPLET` include member. For example, see *orbixhlq*`.DEMOS.IMS.PLI.PLINCL(SIMPLET)`.
10. `OBJREL` is called to ensure that the servant object is released properly.

Location of the SIMPLEC module

You can find a complete version of the IMS `SIMPLEC` client module in *orbixhlq*`.DEMOS.IMS.PLI.SRC(SIMPLEC)`.

Building the Client

JCL to build the client

Sample JCL used to compile and link the client can be found in the third step of *orbixhlq*.DEMOS.IMS.PLI.BLD.JCL(SIMPLEC).

Resulting load module

When the JCL has successfully executed, it results in a load module that is contained in *orbixhlq*.DEMOS.IMS.PLI.LOAD(SIMPLEC).

Preparing the Client to Run in IMS

Overview

This section describes the required steps to allow the client to run in an IMS region. These steps assume you want to run the IMS client against a batch server.

Steps

The steps to enable the client to run in an IMS region are:

Step	Action
1	Define an APPC transaction definition for IMS.
2	Provide the IMS client load module to an IMS region.
3	Start the locator, node daemon, and IFR on the server host.
4	Add the IDL to the IFR.
5	Start the batch server.
6	Customize the batch server IOR.
7	Configure and run the client adapter.

Step 1—Define transaction definition for IMS

A transaction definition must be created for the client, to allow it to run in IMS. The following is the transaction definition for the supplied demonstration:

```
    APPLCTN    GPSB=SIMPLECL,                                      x
          PGMTYPE=(TP,,2),                                         x
          SCHDTYP=PARALLEL
    TRANSACT   CODE=SIMPLECL,                                      x
          EDIT=(ULC)
```

Step 2—Provide client load module to IMS region

Ensure that the *orbixhlq*.DEMOS.IMS.PLI.LOAD PDS is added to the STEPLIB for the IMS region that is to run the transaction.

Note: If you have already done this for your IMS server load module, you do not need to do this again.

Alternatively, you can copy the `SIMPLEC` load module to a PDS in the STEPLIB of the relevant IMS region.

Step 3—Start locator, node daemon, and IFR on server

This step is assuming that you intend running the IMS client against the supplied batch demonstration server.

In this case, you must start all of the following on the batch server host (if they have not already been started):

1. Start the locator daemon by submitting *orbixhlq*`.JCL(LOCATOR)`.
2. Start the node daemon by submitting *orbixhlq*`.JCL(NODEDAEM)`.
3. Start the interface repository by submitting *orbixhlq*`.JCL(IFR)`.

See "Running the Server and Client" on page 44 for more details of running the locator and node daemon on the batch server host.

Step 4—Add IDL to IFR

The client adapter needs to be able to obtain the IDL for the PL/I server from the Interface Repository, so that it knows what data types it can expect to marshal from the IMS transaction, and what data types it should expect back from the batch server. Ensure that the relevant IDL for the server has been added to (that is, registered with) the Interface Repository before the client adapter is started.

To add IDL to the Interface Repository, the IFR server must be running. As explained in "Step 3—Start locator, node daemon, and IFR on server", you can use the JCL in *orbixhlq*`.JCL(IFR)` to start the IFR. The IFR uses the Orbix configuration member for its settings. The IFR uses the configuration settings in the Orbix configuration member, *orbixhlq*`.CONFIG(DEFAULT@)`.

Note: An IDL interface only needs to be registered once with the IFR.

The following JCL that adds IDL to the IFR is supplied in *orbixhlq*.DEMOS.IMS.PLI.BLD.JCL(SIMPLEREG):

```
//          JCLLIB ORDER=(orbixhlq.PROCS)
//          INCLUDE MEMBER=(ORXVARS)
//*
//* Make the following changes before running this JCL:
//*
//* 1.   Change 'SET DOMAIN='DEFAULT@' to your configuration
//*      domain name.
//*
//             SET DOMAIN='DEFAULT@'
//*
//IDLPLI   EXEC ORXIDL,
//          SOURCE=SIMPLE,
//          IDL=&ORBIX..DEMOS.IDL,
//          IDLPARM='-R'
//ITDOMAIN DD DSN=&ORBIX..CONFIG(&DOMAIN),DISP=SHR
```

Step 5—Start batch server

This step is assuming that you intend running the IMS client against the demonstration batch server.

Submit the following JCL to start the batch server:

```
orbixhlq.DEMOS.PLI.RUN.JCL(SIMPLESV)
```

See “Running the Server and Client” on page 44 for more details of running the locator and node daemon on the batch server host.

Step 6—Customize batch server IOR

When you run the demonstration batch server it publishes its IOR to a member called *orbixhlq*.DEMOS.IORS(SIMPLE). The demonstration IMS client needs to use this IOR to contact the demonstration batch server.

The demonstration IMS client obtains the object reference for the demonstration batch server in the form of a corbaloc URL string. A corbaloc URL string can take different formats. For the purposes of this demonstration, it takes the form `corbaloc:rir:/SimpleObject`. This form of the corbaloc URL string requires the use of a configuration variable, `initial_references:SimpleObject:reference`, in the configuration domain. When you submit the JCL in *orbixhlq*.DEMOS.IMS.PLI.BLD.JCL(UPDTCONF), it automatically adds this configuration entry to the configuration domain:

```
initial_references:SimpleObject:reference = "IOR…";
```

The IOR value is taken from the *orbixhlq*.DEMOS.IORS(SIMPLE) member.

See "STR2OBJ" on page 404 for more details of the various forms of corbaloc URL strings and the ways you can use them.

Step 7—Configure and run client adapter

The client adapter must now be configured before you can start the client as a IMS transaction. See the *IMS Adapters Administrator's Guide* for details of how to configure the client adapter.

When you have configured the client adapter, you can run it by submitting the following JCL:

```
orbixhlq.JCL(MFCLA)
```

Running the Demonstrations

Overview

This section provides a summary of what you need to do to successfully run the supplied demonstrations.

In this section

This section discusses the following topics:

Running Batch Client against IMS Server

Overview

This subsection describes what you need to do to successfully run the demonstration batch client against the demonstration IMS server. It also provides an overview of the output produced.

Steps

The steps to run the demonstration IMS server against the demonstration batch client are:

1. Ensure that all the steps in "Preparing the Server to Run in IMS" on page 75 have been successfully completed.
2. Run the batch client as described in "Running the Server and Client" on page 44.

IMS server output

The IMS server sends the following output to the IMS region:

```
Creating the simple_persistent object
Writing out the object reference
Giving control to the ORB to process Requests

Operation call_me() called
```

Batch client output

The batch client produces the following output:

```
simple_persistent demo
=======================
Calling operation call_me…
Operation call_me completed (no results to display)

End of the simple_persistent demo
```

Running IMS Client against Batch Server

Overview

This subsection describes what you need to do to successfully run the demonstration IMS client against the demonstration batch server. It also provides an overview of the output produced.

Steps

The steps to run the demonstration IMS client against the demonstration batch server are:

1. Ensure that all the steps in "Preparing the Client to Run in IMS" on page 85 have been successfully completed.
2. Run the IMS client by entering the transaction name, `SIMPLECL`, in the relevant IMS region.

IMS client output

The IMS client sends the following output to the IMS region:

```
simple_persistent demo
======================
Calling operation call_me…
Operation call_me completed (no results to display)

End of the simple_persistent demo
```

The IMS client sends the following output to the IMS message queue:

```
Simple transaction completed
```

Batch server output

The batch server produces the following output:

```
Creating the simple_persistent object
Writing out the object reference
Giving control to the ORB to process Requests

Operation call_me() called
```

Getting Started in CICS

This chapter introduces CICS application programming with Orbix, by showing how to use Orbix to develop both a CICS PL/I client and a CICS PL/I server. It also provides details of how to subsequently run the CICS client against a PL/I batch server, and how to run a PL/I batch client against the CICS server.

In this chapter

This chapter discusses the following topics:

Note: The client and server examples provided in this chapter respectively require use of the CICS client and server adapters that are supplied as part of the Orbix Mainframe. See the *CICS Adapters Administrator's Guide* for more details about these CICS adapters.

Overview

Introduction

This section provides an overview of the main steps involved in creating an Orbix PL/I CICS server and client application. It also introduces the supplied PL/I CICS client and server `SIMPLE` demonstrations, and outlines where you can find the various source code and JCL elements for them.

Steps to create an application

The main steps to create an Orbix PL/I CICS server application are:

1. "Developing the Application Interfaces" on page 99.
2. "Developing the CICS Server" on page 110.
3. "Developing the CICS Client" on page 122.

For the purposes of illustration this chapter demonstrates how to develop both an Orbix PL/I CICS client and an Orbix PL/I CICS server. It then describes how to run the CICS client and CICS server respectively against a PL/I batch server and a PL/I batch client. These demonstrations do not reflect real-world scenarios requiring the Orbix Mainframe, because the client and server are written in the same language and running on the same platform.

The demonstration CICS server

The Orbix PL/I server developed in this chapter runs in a CICS region. It implements a simple persistent POA-based obect. It accepts and processes requests from an Orbix PL/I batch client that uses the object interface, `SimpleObject`, to communicate with the server via the CICS server adapter. The CICS server uses the Internet Inter-ORB Protocol (IIOP), which runs over TCP/IP, to communicate with the batch client.

The demonstration CICS client

The Orbix PL/I client developed in this chapter runs in a CICS region. It uses the clearly defined object interface, `SimpleObject`, to access and request data from an Orbix PL/I batch server that implements a simple persistent `SimpleObject` object. When the client invokes a remote operation, a request message is sent from the client to the server via the client adapter. When the operation has completed, a reply message is sent back to the client again via the client adapter. The CICS client uses IIOP to communicate with the batch server.

Supplied code and JCL for CICS application development

All the source code and JCL components needed to create and run the CICS `SIMPLE` server and client demonstrations have been provided with your installation. Apart from site-specific changes to some JCL, these do not require editing.

Table 13 provides a summary of these code elements and JCL components (where *orbixhlq* represents your installation's high-level qualifier).

Table 13: *Supplied Code and JCL (Sheet 1 of 2)*

Location	Description
orbixhlq.DEMOS.IDL(SIMPLE)	This is the supplied IDL.
orbixhlq.DEMOS.CICS.PLI.SRC (SIMPLEV)	This is the source code for the CICS server mainline module, which is generated when you run the JCL in *orbixhlq*.DEMOS.CICS.PLI.BLD.JCL(SIMPLIDL). (The CICS server mainline code is not shipped with the product. You must run the SIMPLIDL JCL to generate it.)
orbixhlq.DEMOS.CICS.PLI.SRC (SIMPLEI)	This is the source code for the CICS server implementation module.
orbixhlq.DEMOS.CICS.PLI.SRC (SIMPLEC)	This is the source code for the CICS client module.
orbixhlq.DEMOS.CICS.PLI.BLD.JCL (SIMPLIDL)	This JCL runs the Orbix IDL compiler. See "Orbix IDL Compiler" on page 102 for more details of this JCL and how to use it.
orbixhlq.DEMOS.CICS.PLI.BLD.JCL (SIMPLESB)	This JCL compiles and links the CICS server mainline and CICS server implementation modules to create the SIMPLE server program.
orbixhlq.DEMOS.CICS.PLI.BLD.JCL (SIMPLECB)	This JCL compiles the CICS client module to create the SIMPLE client program.
orbixhlq.DEMOS.CICS.PLI.BLD.JCL (SIMPLREG)	This JCL registers the IDL in the Interface Repository.
orbixhlq.DEMOS.CICS.PLI.BLD.JCL (SIMPLIOR)	This JCL obtains the CICS server's IOR (from the CICS server adapter). A client of the CICS server requires the CICS server's IOR, to locate the server object.

Table 13: *Supplied Code and JCL (Sheet 2 of 2)*

Location	Description
orbixhlq.DEMOS.CICS.PLI.BLD.JCL (UPDTCONF)	This JCL adds the following configuration entry to the configuration member: `initial_references:SimpleObject:reference="IOR…";` This configuration entry specifies the IOR that the CICS client uses to contact the batch server. The IOR that is set as the value for this configuration entry is the IOR that is published in *orbixhlq*.DEMOS.IORS(SIMPLE) when you run the batch server. The object reference for the server is represented to the demonstration CICS client as a corbaloc URL string in the form `corbaloc:rir:/SimpleObject`. This form of corbaloc URL string requires the use of the `initial_references:SimpleObject:reference="IOR…"` configuration entry. Other forms of corbaloc URL string can also be used (for example, the IIOP version, as demonstrated in the nested sequences demonstration supplied with your product installation). See "STR2OBJ" on page 404 for more details of the various forms of corbaloc URL strings and the ways you can use them.
orbixhlq.JCL(MFCLA)	This JCL configures and runs the client adapter.
orbixhlq.JCL(CICSA)	This JCL configures and runs the CICS server adapter.

Supplied include members

Table 14 provides a summary in alphabetic order of the various include members supplied with your product installation that are relevant to CICS application development. Again, *orbixhlq* represents your installation's high-level qualifier.

Table 14: *Supplied Include Members (Sheet 1 of 3)*

Location	Description
orbixhlq.INCLUDE.PLINCL(CHKCLCIC)	This is relevant to CICS clients only. It contains a PL/I function that has been translated via the CICS TS 1.3 translator. This function can be called by the client, to check if a system exception has occurred and report it. It writes any messages raised by the supplied demonstrations to the CICS terminal.

Table 14: *Supplied Include Members (Sheet 2 of 3)*

Location	Description
orbixhlq.INCLUDE.PLINCL(CHKCICS)	This is relevant to CICS clients only. It contains the version of the CHKCLCIC member before it was translated via the CICS TS 1.3 translator. It is used by the CICSTRAN job, to compile the CHKCICS member, using another version of the CICS translator.
orbixhlq.INCLUDE.PLINCL(CHKERRS)	This is relevant to CICS servers. It contains a PL/I function that can be called by the CICS server, to check if a system exception has occurred, and to report that system exception.
orbixhlq.INCLUDE.PLINCL(CORBA)	This is relevant to both CICS clients and servers. It contains common PL/I runtime variables. It includes the CORBACOM include member by default. It also includes the CORBASV include member, if the client module contains the line %client_only='yes';.
orbixhlq.INCLUDE.PLINCL(CORBACOM)	This is relevant to both CICS clients and servers. It contains common PL/I runtime function definitions that can be used both by clients and servers.
orbixhlq.INCLUDE.PLINCL(CORBASV)	This is relevant to CICS servers. It contains PL/I runtime function definitions that can be used by servers.
orbixhlq.INCLUDE.PLINCL(DISPINIT)	This is relevant to CICS servers only. It retrieves the current request information into the REQINFO structure via PODREQ. From REQINFO the operation to be performed by the server is retrieved via a call to STRGET.
orbixhlq.INCLUDE.PLINCL(EXCNAME)	This is relevant to both CICS clients and servers. It contains a PL/I function called CORBA_EXC_NAME that returns the system exception name for the system exception being raised (that is, it maps Orbix exceptions to human-readable strings). EXCNAME is used by CHKERRS and CHKCLCIC.
orbixhlq.INCLUDE.PLINCL(URLSTR)	This is relevant to clients only. It contains a PL/I representation of the corbaloc URL IIOP string format. A client can call STR2OBJ to convert the URL into an object reference. See "STR2OBJ" on page 404 for more details.

Table 14: *Supplied Include Members (Sheet 3 of 3)*

Location	Description
orbixhlq.DEMOS.CICS.PLI.PLINCL	This PDS is relevant to both CICS clients and servers. It is used to store all CICS include members generated when you run the JCL to run the Orbix IDL compiler for the supplied demonstrations. It also contains helper procedures for use with the nested sequences demonstration.
orbixhlq.DEMOS.CICS.MFAMAP	This PDS is relevant to CICS servers only. It is empty at installation time. It is used to store the CICS server adapter mapping member generated when you run the JCL to run the Orbix IDL compiler for the supplied demonstrations. The contents of the mapping member are the fully qualifed interface name followed by the operation name followed by the CICS APPC transaction name or CICS EXCI program name (for example, `(Simple/SimpleObject,call_me,SIMPLESV)`. See the *CICS Adapters Administrator's Guide* for more details about generating CICS server adapter mapping members.

Checking JCL components

When creating either the CICS client or server `SIMPLE` application, check that each step involved within the separate JCL components completes with a condition code not greater than `4`. If the condition codes are greater than `4`, establish the point and cause of failure. The most likely cause is the site-specific JCL changes required for the compilers. Ensure that each high-level qualifier throughout the JCL reflects your installation.

Developing the Application Interfaces

Overview

This section describes the steps you must follow to develop the IDL interfaces for your application. It first describes how to define the IDL interfaces for the objects in your system. It then describes how to run the IDL compiler. Finally it provides an overview of the PL/I include members, server source code, and CICS server adapter mapping member that you can generate via the IDL compiler.

Steps to develop application interfaces

The steps to develop the interfaces to your application are:

Step	Action
1	Define public IDL interfaces to the objects required in your system. See “Defining IDL Interfaces” on page 100.
2	Run the Orbix IDL compiler to generate PL/I include members, server source, and server mapping member. See “Orbix IDL Compiler” on page 102.

Defining IDL Interfaces

Defining the IDL

The first step in writing any Orbix program is to define the IDL interfaces for the objects required in your system. The following is an example of the IDL for the `SimpleObject` interface that is supplied in *orbixhlq*.DEMOS.IDL(SIMPLE):

```
// IDL
module Simple
{
    interface SimpleObject
    {
        void
        call_me();
    };
};
```

Explanation of the IDL

The preceding IDL declares a `SimpleObject` interface that is scoped (that is, contained) within the `Simple` module. This interface exposes a single `call_me()` operation. This IDL definition provides a language-neutral interface to the CORBA `Simple::SimpleObject` type.

How the demonstration uses this IDL

For the purposes of the demonstrations in this chapter, the `SimpleObject` CORBA object is implemented in PL/I in the supplied simple server application. The server application creates a persistent server object of the `SimpleObject` type, and publishes its object reference to a PDS member. The client invokes the `call_me()` operation on the `SimpleObject` object, and then exits.

The batch demonstration client of the CICS demonstration server locates the `SimpleObject` object by reading the interoperable object reference (IOR) for the CICS server adapter from *orbixhlq*.DEMOS.IORS(SIMPLE). In this case, the CICS server adapter IOR is published to *orbixhlq*.DEMOS.IORS(SIMPLE) when you run *orbixhlq*.DEMOS.CICS.PLI.BLD.JCL(SIMPLIOR).

The CICS demonstration client of the batch demonstration server locates the `SimpleObject` object by reading the IOR for the batch server from *orbixhlq*.DEMOS.IORS(SIMPLE). In this case, the batch server IOR is published to *orbixhlq*.DEMOS.IORS(SIMPLE) when you run the batch server. The object reference for the server is represented to the demonstration CICS client as a corbaloc URL string in the form corbaloc:rir:/SimpleObject.

Orbix IDL Compiler

The Orbix IDL compiler

This subsection describes how to use the Orbix IDL compiler to generate PL/I include members, server source, and the CICS server adapter mapping member from IDL.

Note: Generation of PL/I include members is relevant to both CICS client and server development. Generation of server source and the CICS server adapter mapping member is relevant only to CICS server development.

Orbix IDL compiler configuration

The Orbix IDL compiler uses the Orbix configuration member for its settings. The `SIMPLIDL` JCL that runs the compiler uses the configuration member *orbixhlq*`.CONFIG(IDL)`. See "Orbix IDL Compiler" on page 233 for more details.

Example of the SIMPLIDL JCL

The following JCL runs the IDL compiler for the CICS `SIMPLE` demonstration:

```
//SIMPLIDL JOB   (),
//         CLASS=A,
//         MSGCLASS=X,
//         MSGLEVEL=(1,1),
//         REGION=0M,
//         TIME=1440,
//         NOTIFY=&SYSUID,
//         COND=(4,LT)
//*----------------------------------------------------------------
//* Orbix - Generate PL/I CICS server files for the Simple Demo
//*----------------------------------------------------------------
//         JCLLIB ORDER=(orbixhlq.PROCS)
//         INCLUDE MEMBER=(ORXVARS)
//*
//* Make the following changes before running this JCL:
//*
//* 1.   Change 'SET DOMAIN='DEFAULT@' to your configuration
//*      domain name.
//*
//            SET DOMAIN='DEFAULT@'
//*
```

```
//IDLPLI   EXEC ORXIDL,
//         SOURCE=SIMPLE,
//         IDL=&ORBIX..DEMOS.IDL,
//         IDLPARM='-pli:-TCICS -mfa:-tSIMPLESV'
//*        IDLPARM='-pli:-TCICS -mfa:-tSMSV'
//*        IDLPARM='-pli:-V'
//IDLMFA   DD DISP=SHR,DSN=&ORBIX..DEMOS.CICS.MFAMAP
//ITDOMAIN DD DSN=&ORBIX..CONFIG(&DOMAIN),DISP=SHR
```

Explanation of the SIMPLIDL JCL

In the preceding JCL example, the `IDLPARM` lines can be explained as follows:

- The line `IDLPARM='-pli:-TCICS -mfa:-tSIMPLESV'` is relevant to CICS server development for EXCI. This line generates:
    - PL/I include members via the `-pli` argument.
    - CICS server mainline code via the `-TCICS` arguments.
    - CICS server adapter mapping member via the `-mfa:-t`*`tran_or_program_name`* arguments.

    Note: Because CICS server implementation code is already supplied for you, the `-S` argument is not specified by default.

- The line `IDLPARM='-pli:-TCICS -mfa:-tSMSV'` is relevant to CICS server development for APPC. This line generates the same items as the `IDLPARM='-pli:-TCICS -mfa:-tSIMPLESV'`. It is disabled (that is, commented out with an asterisk) by default.
- The line `IDLPARM='-pli:-V'` is relevant to CICS client development and generates only PL/I include members, because it only specifies the `-pli:-V` arguments. (The `-V` argument prevents generation of PL/I server mainline source code.) It is disabled (that is, commented out) by default.

    Note: The Orbix IDL compiler does not generate PL/I client source code.

For the purposes of the demonstration, the IDLPARM='-pli:-TCICS -mfa:-tSIMPLESV' line is not commented out (that is, it is not preceded by an asterisk) by default.

Specifying what you want to generate

To indicate which one of the IDLPARM lines you want SIMPLIDL to recognize, comment out the two IDLPARM lines you do not want to use, by ensuring an asterisk precedes those lines. By default, as shown in the preceding example, the JCL is set to generate PL/I include members, server mainline code, and a CICS server adapter mapping member for EXCI.

See "Orbix IDL Compiler" on page 233 for more details of the Orbix IDL compiler and the JCL used to run it.

Running the Orbix IDL compiler

After you have edited the SIMPLIDL JCL according to your requirements, you can run the Orbix IDL compiler by submitting the following job:

```
orbixhlq.DEMOS.CICS.PLI.BLD.JCL(SIMPLIDL)
```

Generated PL/I Include Members, Source, and Mapping Member

Overview

This subsection describes all the PL/I include members, server source, and CICS server adapter mapping member that the Orbix IDL compiler can generate from IDL definitions.

> **Note:** The generated PL/I include members are relevant to both CICS client and server development. The generated source and adapter mapping member are relevant only to CICS server development. The IDL compiler does not generate PL/I client source.

Member name restrictions

Generated PL/I source code, include, and mapping member names are all based on the IDL member name. If the IDL member name exceeds six characters, the Orbix IDL compiler uses only the first six characters of the IDL member name when generating the other member names. This allows space for appending a one-character suffix to each generated member name, while allowing it to adhere to the seven-character maximum size limit for PL/I external procedure names, which are based by default on the generated member names.

How IDL maps to PL/I include members

Each IDL interface maps to a set of PL/I structures. There is one structure defined for each IDL operation. A structure contains each of the parameters for the relevant IDL operation in their corresponding PL/I representation. See "IDL-to-PL/I Mapping" on page 177 for details of how IDL types map to PL/I.

Attributes map to two operations (`get` and `set`), and readonly attributes map to a single `get` operation.

Generated PL/I include members

Table 15 shows the PL/I include members that the Orbix IDL compiler generates, based on the defined IDL..

Table 15: *Generated PL/I Include Members (Sheet 1 of 2)*

Copybook	JCL Keyword Parameter	Description
*idlmembername*D	COPYLIB	This include member contains a select statement that determines which server implementation procedure is to be called, based on the interface name and operation received.
*idlmembername*L	COPYLIB	This include member contains structures and procedures used by the PL/I runtime to read and store data into the operation parameters. This member is automatically included in the *idlmembername*X include member.
*idlmembername*M	COPYLIB	This include member contains declarations and structures that are used for working with operation parameters and return values for each interface defined in the IDL member. The structures use the based PL/I structures declared in the *idlmembername*T include member. This member is automatically included in the *idlmembername*I include member.

Table 15: *Generated PL/I Include Members (Sheet 2 of 2)*

Copybook	JCL Keyword Parameter	Description
*idlmembername*T	COPYLIB	This include member contains the based structure declarations that are used in the *idlmembername*M include member. This member is automatically included in the *idlmembername*M include member.
*idlmembername*X	COPYLIB	This include member contains structures that are used by the PL/I runtime to support the interfaces defined in the IDL member. This member is automatically included in the *idlmembername*V source code member.
*idlmembername*D	COPYLIB	This include member contains a select statement for calling the correct procedure for the requested operation. This include member is automatically included in the *idlmembername*I source code member.

Generated server source members

Table 16 shows the server source code members that the Orbix IDL compiler generates, based on the defined IDL.:

Table 16: *Generated Server Source Code Members*

Member	JCL Keyword Parameter	Description
*idlmembername*I	IMPL	This is the CICS server implementation source code member. It contains procedure definitions for all the callable operations. The is only generated if you specify both the -S and -TCICS arguments with the IDL compiler.
*idlmembername*V	IMPL	This is the CICS server mainline source code member. It is generated by default. However, you can use the -V argument with the IDL compiler, to prevent generation of this member.

Note: For the purposes of this example, the SIMPLEI server implementation member is already provided in your product installation. Therefore, the -S IDL compiler argument used to generate it is not specified in the supplied SIMPLIDL JCL. The SIMPLEV server mainline member is not already provided, so the -V argument, which prevents generation of server mainline code, is not specified in the supplied JCL. See "Orbix IDL Compiler" on page 233 for more details of the IDL compiler arguments used to generate, and prevent generation of, CICS server source code.

Generated server adapter mapping member

Table 17 shows the CICS server adapter mapping member that the Orbix IDL compiler generates, based on the defined IDL.

Table 17: *Generated CICS Server Adapter Mapping Member*

Copybook	JCL Keyword Parameter	Description
`*idlmembername*A`	`MEMBER`	This is a simple text file that determines what interfaces and operations the CICS server adapter supports, and the CICS APPC transaction names, or CICS EXCI program names, to which the CICS server adapter should map each IDL operation.

Location of demonstration include and mapping member

You can find examples of the include members, server source, and CICS server adapter mapping member generated for the `SIMPLE` demonstration in the following locations:

- *orbixhlq*.DEMOS.CICS.PLI.PLINCL(SIMPLED)
- *orbixhlq*.DEMOS.CICS.PLI.PLINCL(SIMPLEL)
- *orbixhlq*.DEMOS.CICS.PLI.PLINCL(SIMPLEM)
- *orbixhlq*.DEMOS.CICS.PLI.PLINCL(SIMPLET)
- *orbixhlq*.DEMOS.CICS.PLI.PLINCL(SIMPLEX)
- *orbixhlq*.DEMOS.CICS.PLI.SRC(SIMPLEV)
- *orbixhlq*.DEMOS.CICS.PLI.SRC(SIMPLEI)
- *orbixhlq*.DEMOS.CICS.MFAMAP(SIMPLEA)

Note: Except for the `SIMPLEI` member, none of the preceding elements are shipped with your product installation. They are generated when you run *orbixhlq*.DEMOS.CICS.PLI.BLD.JCL(SIMPLIDL), to run the Orbix IDL compiler.

Developing the CICS Server

Overview

This section describes the steps you must follow to develop the CICS server executable for your application. The CICS server developed in this example will be contacted by the simple batch client demonstration.

Steps to develop the server

The steps to develop the server application are:

Step	Action
1	"Writing the Server Implementation" on page 111.
2	"Writing the Server Mainline" on page 114.
3	"Building the Server" on page 117.
4	"Preparing the Server to Run in CICS" on page 118.

Writing the Server Implementation

The server implementation module

You must implement the server interface by writing a PL/I implementation module that implements each operation defined to the operation section in the *idlmembername*T include member. For the purposes of this example, you must write a PL/I procedure that implements each operation in the SIMPLET include member. When you specify the -S and -TCICS arguments with the Orbix IDL compiler, it generates a skeleton server implementation module, in this case called SIMPLEI, which is a useful starting point.

Note: For the purposes of this demonstration, the CICS server implementation module, SIMPLEI, is already provided for you, so the -S argument is not specified in the JCL that runs the IDL compiler.

Example of the CICS SIMPLEI module

The following is an example of the CICS SIMPLEI module (with the header comment block omitted for the sake of brevity):

Example 4: *The SIMPLEI Demonstration Module (Sheet 1 of 2)*

```
   SIMPLEI: PROC;

   /*The following line enables the runtime to call this procedure*/
1  DISPTCH: ENTRY;

   dcl (addr,low,sysnull)                  builtin;

   %include CORBA;
   %include CHKERRS;
   %include SIMPLEM;
2  %include DISPINIT;

   /* ================ Start of global user code =============== */
   /* ================= End of global user code ================ */

   /* ----------------------------------------------------------*/
   /*                                                           */
   /* Dispatcher : select(operation)                            */
   /*                                                           */
   /*-----------------------------------------------------------*/
3  %include SIMPLED;
```

Example 4: *The SIMPLEI Demonstration Module (Sheet 2 of 2)*

```
  /*------------------------------------------------------------*/
  /* Interface:                                                 */
  /*     Simple/SimpleObject                                    */
  /*                                                            */
  /* Mapped name:                                               */
  /*     Simple_SimpleObject                                    */
  /*                                                            */
  /* Inherits interfaces:                                       */
  /*     (none)                                                 */
  /*------------------------------------------------------------*/
  /*------------------------------------------------------------*/
  /* Operation:        call_me                                  */
  /* Mapped name:      call_me                                  */
  /* Arguments:        None                                     */
  /* Returns:          void                                     */
  /*------------------------------------------------------------*/
4 proc_Simple_SimpleObject_c_c904: PROC(p_args);

  dcl p_args                  ptr;
5 dcl 1_args                  aligned based(p_args)
                              like Simple_SimpleObject_c_ba77_type;

  /* ==================== Start of operation code ============= */
6 put skip list('Operation call_me() called');
  put skip;
  /* ===================== End of operation code ============== */

  END proc_Simple_SimpleObject_c_c904;

  END SIMPLEI;
```

Explanation of the CICS SIMPLEI module

The CICS `SIMPLEI` module can be explained as follows:

1. When an incoming request arrives from the network, it is processed by the ORB and a call is made from the PL/I runtime to the `DISPTCH` entry point.
2. Within the `DISPINIT` include member, `PODREQ` is called to provide information about the current invocation request, which is held in the `REQINFO` structure. `PODREQ` is called once for each operation invocation after a request has been dispatched to the server. `STRGET` is then called to copy the characters in the unbounded string pointer for the operation name into the PL/I string that represents the operation name.

3. The `SIMPLED` include member contains a select statement that determines which procedure within `SIMPLEI` is to be called, given the operation name and interface name passed to `SIMPLEI`. It calls `PODGET` before the call to the server procedure, which fills the appropriate PL/I structure declared in the main include member, `SIMPLEM`, with the operation's incoming arguments. It then calls `PODPUT` after the call to the server procedure, to send out the operation's outgoing arguments.
4. The procedural code containing the server implementation for the `call_me` operation.
5. Each operation has an argument structure and these are declared in the typecode include member, `SIMPLET`. If an operation does not have any parameters or return type, such as `call_me`, the structure only contains a structure with a dummy char.
6. This is a sample of the server implementation code for `call_me`. It is the only part of the `SIMPLEI` member that is not automatically generated by the Orbix IDL compiler.

Note: An operation implementation should not call `PODGET` or `PODPUT`. These calls are made within the `SIMPLED` include member generated by the Orbix IDL compiler.

Location of the CICS SIMPLEI module

You can find a complete version of the CICS `SIMPLEI` server implementation module in *`orbixhlq`*`.DEMOS.CICS.PLI.SRC(SIMPLEI)`.

Writing the Server Mainline

The server mainline module

The next step is to write the server mainline module in which to run the server implementation. For the purposes of this example, when you specify the `-TCICS` argument with the Orbix IDL compiler, it generates a module called `SIMPLEV`, which contains the server mainline code.

Note: Unlike the batch server mainline, the CICS server mainline does not have to create and store stringified object references (IORs) for the interfaces that it implements, because this is handled by the CICS server adapter.

Example of the CICS SIMPLEV module

The following is an example of the CICS `SIMPLEV` module:

Example 5: *The SIMPLEV Demonstration Module (Sheet 1 of 2)*

```
SIMPLEV: PROC OPTIONS(MAIN NOEXECOPS);

dcl arg_list                     char(01)        init('');
dcl arg_list_len                 fixed bin(31)   init(0);
dcl orb_name                    char(10)        init('simple_orb');
dcl orb_name_len                 fixed bin(31)   init(10);
dcl srv_name                     char(256) var;
dcl server_name                  char(07)        init('simple ');
dcl server_name_len              fixed bin(31)   init(6);

dcl Simple_SimpleObject_objid  char(27)
                            init('Simple/SimpleObject_object ');
dcl Simple_SimpleObject_obj    ptr;
dcl SYSPRINT                     file stream output;
dcl (addr,length,low,sysnull)  builtin;

%include CORBA;
%include CHKERRS;
%include SIMPLET;
%include SIMPLEX;

alloc pod_status_information set(pod_status_ptr);

1  call podstat(pod_status_ptr);
```

Example 5: *The SIMPLEV Demonstration Module (Sheet 2 of 2)*

```
  if check_errors('podstat') ^= completion_status_yes then return;

  /* Initialize the server connection to the ORB                    */
2 call orbargs(arg_list,arg_list_len,orb_name,orb_name_len);
  if check_errors('orbargs') ^= completion_status_yes then return;

3 call podsrvr(server_name, server_name_len);
  if check_errors('podsrvr') ^= completion_status_yes then return;

  /* Register interface : Simple/SimpleObject                       */
4 call podreg(addr(Simple_SimpleObject_interface));
  if check_errors('podreg';) ^= completion_status_yes then return;

5 call objnew(server_name,
              Simple_SimpleObject_intf,
              Simple_SimpleObject_objid,
              Simple_SimpleObject_obj);
  if check_errors('objnew') ^= completion_status_yes then return;

  /* Server is now ready to accept requests                         */
6 call podrun;
  if check_errors('podrun') ^= completion_status_yes then return;

7 call objrel(Simple_SimpleObject_obj);
  if check_errors('objrel') ^= completion_status_yes then return;

  free pod_status_information;

  END SIMPLEV;
```

Explanation of the CICS SIMPLEV module

The CICS `SIMPLEV` module can be explained as follows:

1. `PODSTAT` is called to register the `POD_STATUS_INFORMATION` block that is contained in the `CORBA` include member. Registering the `POD_STATUS_INFORMATION` block allows the PL/I runtime to populate it with exception information, if necessary. If `completion_status` is set to zero after a call to the PL/I runtime, this means that the call has completed successfully.
2. `ORBARGS` is called to initialize a connection to the ORB.
3. `PODSRVR` is called to set the server name.

4. `PODREG` is called to register the IDL interface, `SimpleObject`, with the PL/I runtime.
5. `OBJNEW` is called to create a persistent server object of the `SimpleObject` type, with an object ID of `my_simple_object`.
6. `PODRUN` is called, to enter the `ORB::run()` loop, to allow the ORB to receive and process client requests. This then processes the CORBA request that the CICS adapter sends to CICS.
7. `OBJREL` is called to ensure that the servant object is released properly.

See the preface of this guide for details about the compilers that this product supports.

Location of the CICS SIMPLEV module

You can find a complete version of the CICS `SIMPLEV` server mainline module in *`orbixhlq`*`.DEMOS.CICS.PLI.SRC(SIMPLEV)` after you have run *`orbixhlq`*`.DEMOS.CICS.PLI.BLD.JCL(SIMPLIDL)` to run the Orbix IDL compiler.

Building the Server

Location of the JCL

Sample JCL used to compile and link the CICS server mainline and server implementation is in *orbixhlq*.DEMOS.CICS.PLI.BLD.JCL(SIMPLESB).

Resulting load module

When this JCL has successfully executed, it results in a load module that is contained in *orbixhlq*.DEMOS.CICS.PLI.LOAD(SIMPLESV).

Preparing the Server to Run in CICS

Overview

This section describes the required steps to allow the server to run in a CICS region. These steps assume you want to run the CICS server against a batch client. When all the steps in this section have been completed, the server is started automatically within CICS, as required.

Steps

The steps to enable the server to run in a CICS region are:

Step	Action
1	Define an APPC transaction definition or EXCI program definition for CICS.
2	Provide the CICS server load module to a CICS region.
3	Generate mapping member entries for the CICS server adapter.
4	Add the IDL to the Interface Repository (IFR). **Note:** For the purposes of this demonstration, the IFR is used as the source of type information.
5	Obtain the IOR for use by the client program.

Step 1—Defining program or transaction definition for CICS

A CICS APPC transaction definition, or CICS EXCI program definition, must be created for the server, to allow it to run in CICS. The following is the CICS APPC transaction definition for the supplied demonstration:

```
DEFINE  TRANSACTION(SMSV)
        GROUP(ORXAPPC)
        DESCRIPTION(Orbix APPC Simple demo transaction)
        PROGRAM(SIMPLESV)
        PROFILE(DFHCICSA)
        TRANCLASS(DFHTCL00)
        DTIMOUT(10)
        SPURGE(YES)
        TPURGE(YES)
        RESSEC(YES)
```

The following is the CICS EXCI program definition for the supplied demonstration:

```
DEFINE  PROGRAM(SIMPLESV)
        GROUP(ORXDEMO)
        DESCRIPTION(Orbix Simple demo server)
        LANGUAGE(LE370)
        DATALOCATION(ANY)
        EXECUTIONSET(DPLSUBSET)
```

See the supplied *orbixhlq*.JCL(ORBIXCSD) for a more detailed example of how to define the resources that are required to use Orbix with CICS and to run the supplied demonstrations.

Step 2—Providing load module to CICS region

Ensure that the *orbixhlq*.DEMOS.CICS.PLI.LOAD PDS is added to the DFHRPL for the CICS region that is to run the transaction, or copy the SIMPLESV load module to a PDS in the DFHRPL of the relevant CICS region.

Step 3—Generating mapping member entries

The CICS server adapter requires mapping member entries, so that it knows which CICS APPC transaction or CICS EXCI program should be run for a particular interface and operation. The mapping member entry for the supplied CICS EXCI server example is contained by default in *orbixhlq*.DEMOS.CICS.MFAMAP(SIMPLEA) after you run the IDL compiler. The mapping member entry for EXCI appears as follows:

```
(Simple/SimpleObject,call_me,SIMPLESV)
```

> **Note:** If instead you chose to enable the line in SIMPLIDL to generate a mapping member entry for a CICS APPC version of the demonstration, that mapping member entry would appear as follows:
>
> (Simple/SimpleObject,call_me,SMSV)

The generation of a mapping member for the CICS server adapter is performed by the *orbixhlq*.DEMOS.CICS.PLI.BLD.JCL(SIMPLIDL) JCL. The -mfa:-t*tran_or_program_name* argument with the IDL compiler generates the mapping member. For the purposes of this example, *tran_or_program_name* is replaced with SIMPLESV. An IDLMFA DD statement must also be provided in the JCL, to specify the PDS into which the mapping member is generated. See the *CICS Adapters Administrator's Guide* for full details about CICS adapter mapping members.

Step 4—Adding IDL to Interface Repository

The CICS server adapter needs to be able to obtain operation signatures for the PL/I server. For the purposes of this demonstration, the IFR is used to retrieve this type information. This type information is necessary so that the adapter knows what data types it has to marshal into CICS for the server, and what data types it can expect back from the CICS APPC transaction or CICS EXCI program. Ensure that the relevant IDL for the server has been added to (that is, registered with) the Interface Repository before the CICS server adapter is started.

To add IDL to the Interface Repository, the Interface Repository must be running. You can use the JCL in *orbixhlq*.JCL(IFR) to start it. The Interface Repository uses the configuration settings in the Orbix configuration member, *orbixhlq*.CONFIG(DEFAULT@).

The following JCL that adds IDL to the Interface Repository is supplied in *orbixhlq*.DEMOS.CICS.PLI.BLD.JCL(SIMPLEREG):

```
//         JCLLIB ORDER=(orbixhlq.PROCS)
//         INCLUDE MEMBER=(ORXVARS)
//*
//* Make the following changes before running this JCL:
//*
//* 1.   Change 'SET DOMAIN='DEFAULT@' to your configuration
//*      domain name.
//*
//            SET DOMAIN='DEFAULT@'
//*
//IDLPLI   EXEC ORXIDL,
//         SOURCE=SIMPLE,
//         IDL=&ORBIX..DEMOS.IDL,
//         IDLPARM='-R'
//ITDOMAIN DD DSN=&ORBIX..CONFIG(&DOMAIN),DISP=SHR
```

Note: An alternative to using the IFR is to use type information files. These are an alternative method of providing IDL interface information to the CICS server adapter. Type information files can be generated as part of the `-mfa` plug-in to the IDL compiler. See the *CICS Adapters Administrator's Guide* for more details about how to generate them. The use of type information files would render this step unnecessary; however, the use of the IFR is recommended for the purposes of this demonstration.

Step 5—Obtaining the server adapter IOR

The final step is to obtain the IOR that the batch client needs to locate the CICS server adapter. Before you do this, ensure all of the following:

- The IFR server is running and contains the relevant IDL. See "Step 4—Adding IDL to Interface Repository" on page 120 for details of how to start it, if it is not already running.
- The CICS server adapter is running. The supplied JCL in *orbixhlq*.JCL(CICSA) starts the CCIS server adapter. See the *CICS Adapters Administrator's Guide* for more details.
- The CICS server adapter mapping member contains the relevant mapping entries. For the purposes of this example, ensure that the *orbixhlq*.DEMOS.CICS.MFAMAP(SIMPLEA) mapping member is being used. See the *CICS Adapters Administrator's Guide* for details about CICS server adapter mapping members.

Now submit *orbixhlq*.DEMOS.CICS.PLI.BLD.JCL(SIMPLIOR), to obtain the IOR that the batch client needs to locate the CICS server adapter. This JCL includes the `resolve` command, to obtain the IOR. The following is an example of the `SIMPLIOR` JCL:

```
//          JCLLIB ORDER=(orbixhlq.PROCS)
//          INCLUDE MEMBER=(ORXVARS)
//*
//* Request the IOR for the CICS 'simple_persistent' server
//* and store it in a PDS for use by the client.
//*
//* Make the following changes before running this JCL:
//*
//* 1.   Change 'SET DOMAIN='DEFAULT@' to your configuration
//*      domain name.
//*
//             SET DOMAIN='DEFAULT@'
//*
//REG      EXEC PROC=ORXADMIN,
//  PPARM='mfa resolve Simple/SimpleObject > DD:IOR'
//IOR DD DSN=&ORBIX..DEMOS.IORS(SIMPLE),DISP=SHR
//ORBARGS DD *
-ORBname iona_utilities.cicsa
/*
//ITDOMAIN DD DSN=&ORBIX..CONFIG(&DOMAIN),DISP=SHR
```

Developing the CICS Client

Overview

This section describes the steps you must follow to develop the CICS client executable for your application. The CICS client developed in this example will connect to the simple batch server demonstration.

Note: The Orbix IDL compiler does not generate PL/I client stub code.

Steps to develop the client

The steps to develop and run the client application are:

Step	Action
1	"Writing the Client" on page 123.
2	"Building the Client" on page 127.
3	"Preparing the Client to Run in CICS" on page 128.

Writing the Client

The client module

The next step is to write the client module, to implement the CICS client. This example uses the supplied `SIMPLECL` client demonstration.

Example of the SIMPLEC module

The following is an example of the CICS `SIMPLEC` module:

Example 6: *The SIMPLEC Demonstration Module (Sheet 1 of 3)*

```
SIMPLEC: PROC OPTIONS(MAIN NOEXECOPS);
 %client_only='yes';

 dcl (addr,substr,sysnull,low,length)   builtin;

 dcl arg_list                        char(40)       init('');
 dcl arg_list_len                    fixed bin(31)  init(38);
 dcl orb_name                        char(10)
                                     init('simple_orb');
 dcl orb_name_len                    fixed bin(31)  init(10);

 dcl sysprint                        file stream output;

1 dcl simple_url               char(27)
                         init('corbaloc:rir:/SimpleObject ');
 dcl simple_url_ptr                  ptr  init(sysnull());
 dcl Simple_SimpleObject_obj         ptr;

 dcl MessageText                     char(79)       init('');

 %include CORBA;
 %include CHKCLCIC;
 %include SIMPLEM;
 %include SIMPLEX;

 /* Initialize the PL/I runtime status information block */
 alloc pod_status_information set(pod_status_ptr);
2 call podstat(pod_status_ptr);

 /* Initialize our ORB */
3 call orbargs(arg_list,
              arg_list_len,
              orb_name,
              orb_name_len);
```

Example 6: *The SIMPLEC Demonstration Module (Sheet 2 of 3)*

```
   if check_errors('orbargs') ^= completion_status_yes then
      exec cics return;

   /* Register the SimpleObject intf with the PL/I runtime */
4  call podreg(addr(Simple_SimpleObject_interface));
   if check_errors('podreg') ^= completion_status_yes then
      exec cics return;

   /* Create an object reference from the server's URL */
   /* so we can make calls to the server               */
5  call strset(simple_url_ptr,
               simple_url,
               length(simple_url));


   if check_errors('strset') ^= completion_status_yes then
      exec cics return;

6  call str2obj(simple_url_ptr,Simple_SimpleObject_obj);
   if check_errors('str2obj') ^= completion_status_yes then
      exec cics return;

   /* Now we are ready to start making server requests */

   put skip list('simple_persistent demo');
   put skip list('======================');

   /* Call operation call_me */
   put skip list('Calling operation call_me...');
7  call podexec(Simple_SimpleObject_obj,
                Simple_SimpleObject_call_me,
                addr(Simple_SimpleObject_c_ba77_args),
                no_user_exceptions);
   if check_errors('podexec') ^= completion_status_yes then
      exec cics return;

   put skip list('Operation call_me completed (no results to
     display)');
   put skip;
   put skip list('End of the simple_persistent demo');
   put skip;

   MessageText = 'Simple Transaction completed';

8  EXEC CICS SEND TEXT FROM (MessageText) LENGTH(79) FREEKB;
```

Example 6: *The SIMPLEC Demonstration Module (Sheet 3 of 3)*

```
    /* Free the simple_persistent object reference */
9   call objrel(Simple_SimpleObject_obj);
    if check_errors('objrel') ^= completion_status_yes then
       exec cics return;

    free pod_status_information;
    exec cics return;

    END SIMPLEC;
```

Explanation of the SIMPLEC module

The CICS `SIMPLEC` module can be explained as follows:

1. `simple_url` defines a corbaloc URL string in the `corbaloc:rir` format. This string identifies the server with which the client is to communicate. This string can be passed as a parameter to `STR2OBJ`, to allow the client to retrieve an object reference to the server. See point **6** about `STR2OBJ` for more details.
2. `PODSTAT` is called to register the `POD_STATUS_INFORMATION` block that is contained in the `CORBA` include member. Registering the `POD_STATUS_INFORMATION` block allows the PL/I runtime to populate it with exception information, if necessary. If `completion_status` is set to zero after a call to the PL/I runtime, this means that the call has completed successfully.

   The `check_errors` function can be used to test the status of any Orbix call. It tests the value of the `exception_number` in `pod_status_information`. If its value is zero, it means the call was successful. Otherwise, `check_errors` prints out the system exception number and message, and the program ends at that point. The `check_errors` function should be called after every PL/I runtime call to ensure the call completed successfully.
3. `ORBARGS` is called to initialize a connection to the ORB.
4. `PODREG` is called to register the IDL interface with the Orbix PL/I runtime.
5. `STRSET` is called to create an unbounded string to which the stringified object reference is copied.

6. `STR2OBJ` is called to create an object reference to the server object. This must be done to allow operation invocations on the server. In this case, the client identifies the target object, using a corbaloc URL string in the form `corbaloc:rir:/SimpleObject` (as defined in point **1**). See "STR2OBJ" on page 404 for more details of the various forms of corbaloc URL strings and the ways you can use them.
7. After the object reference is created, `PODEXEC` is called to invoke operations on the server object represented by that object reference. You must pass the object reference, the operation name, the argument description packet, and the user exception buffer. If the call does not have a user exception defined (as in the preceding example), the `no_user_exceptions` variable is passed in instead. The operation name must be terminated with a space. The same argument description is used by the server. For ease of use, string identifiers for operations are defined in the `SIMPLET` include member. For example, see *orbixhlq*`.DEMOS.CICS.PLI.PLINCL(SIMPLET)`.
8. The `EXEC CICS SEND` statement is used to write messages to the CICS terminal. The client uses this to indicate whether the call was successful or not.
9. `OBJREL` is called to ensure that the servant object is released properly.

Location of the SIMPLEC module

You can find a complete version of the CICS `SIMPLEC` client module in *orbixhlq*`.DEMOS.CICS.PLI.SRC(SIMPLEC)`.

Building the Client

JCL to build the client

Sample JCL used to compile and link the client can be found in the third step of *orbixhlq*.DEMOS.CICS.PLI.BLD.JCL(SIMPLECB).

Resulting load module

When the JCL has successfully executed, it results in a load module that is contained in *orbixhlq*.DEMOS.CICS.PLI.LOAD(SIMPLECL).

Preparing the Client to Run in CICS

Overview

This section describes the required steps to allow the client to run in a CICS region. These steps assume you want to run the CICS client against a batch server.

Steps

The steps to enable the client to run in a CICS region are:

Step	Action
1	Define an APPC transaction definition for CICS.
2	Provide the CICS client load module to a CICS region.
3	Start the locator, node daemon, and IFR on the server host.
4	Add the IDL to the IFR.
5	Start the batch server.
6	Customize the batch server IOR.
7	Configure and run the client adapter.

Step 1—Define transaction definition for CICS

A CICS APPC transaction definition must be created for the client, to allow it to run in CICS. The following is the CICS APPC transaction definition for the supplied demonstration:

```
DEFINE  TRANSACTION(SMCL)
        GROUP(ORXDEMO)
        DESCRIPTION(Orbix Client Simple demo transaction)
        PROGRAM(SIMPLECL)
        PROFILE(DFHCICSA)
        TRANCLASS(DFHTCL00)
        DTIMOUT(10)
        SPURGE(YES)
        TPURGE(YES)
        RESSEC(YES)
```

See the supplied *orbixhlq*.JCL(ORBIXCSD) for a more detailed example of how to define the resources that are required to use Orbix with CICS and to run the supplied demonstrations.

Step 2—Provide client load module to CICS region

Ensure that the *orbixhlq*.DEMOS.CICS.PLI.LOAD PDS is added to the DFHRPL for the CICS region that is to run the transaction.

Note: If you have already done this for your CICS server load module, you do not need to do this again.

Alternatively, you can copy the SIMPLECL load module to a PDS in the DFHRPL of the relevant CICS region.

Step 3—Start locator, node daemon, and IFR on server

This step is assuming that you intend running the CICS client against the supplied batch demonstration server.

In this case, you must start all of the following on the batch server host (if they have not already been started):

1. Start the locator daemon by submitting *orbixhlq*.JCL(LOCATOR).
2. Start the node daemon by submitting *orbixhlq*.JCL(NODEDAEM).
3. Start the interface repository by submitting *orbixhlq*.JCL(IFR).

See “Running the Server and Client” on page 44 for more details of running the locator and node daemon on the batch server host.

Step 4—Add IDL to IFR

The client adapter needs to be able to obtain the IDL for the PL/I server from the Interface Repository, so that it knows what data types it can expect to marshal from the CICS APPC transaction, and what data types it should expect back from the batch server. Ensure that the relevant IDL for the server has been added to (that is, registered with) the Interface Repository before the client adapter is started.

To add IDL to the Interface Repository, the Interface Repository must be running. As explained in “Step 3—Start locator, node daemon, and IFR on server”, you can use the JCL in *orbixhlq*.JCL(IFR) to start the IFR. The IFR uses the Orbix configuration member for its settings. The Interface Repository uses the configuration settings in the Orbix configuration member, *orbixhlq*.CONFIG(DEFAULT@).

Note: An IDL interface only needs to be registered once with the Interface Repository.

The following JCL that adds IDL to the Interface Repository is supplied in *orbixhlq*.DEMOS.CICS.PLI.BLD.JCL(SIMPLEREG):

```
//         JCLLIB ORDER=(orbixhlq.PROCS)
//         INCLUDE MEMBER=(ORXVARS)
//*
//* Make the following changes before running this JCL:
//*
//* 1.  Change 'SET DOMAIN='DEFAULT@' to your configuration
//*     domain name.
//*
//           SET DOMAIN='DEFAULT@'
//*
//IDLPLI   EXEC ORXIDL,
//         SOURCE=SIMPLE,
//         IDL=&ORBIX..DEMOS.IDL,
//         IDLPARM='-R'
//ITDOMAIN DD DSN=&ORBIX..CONFIG(&DOMAIN),DISP=SHR
```

Step 5—Start batch server

This step is assuming that you intend running the CICS client against the demonstration batch server.

Submit the following JCL to start the batch server:

```
orbixhlq.DEMOS.PLI.RUN.JCL(SIMPLESV)
```

See “Running the Server and Client” on page 44 for more details of running the locator and node daemon on the batch server host.

Step 6—Customize batch server IOR

When you run the batch server it publishes its IOR to a member called *orbixhlq*.DEMOS.IORS(SIMPLE). The CICS client needs to use this IOR to contact the server.

The demonstration CICS client obtains the object reference for the demonstration batch server in the form of a corbaloc URL string. A corbaloc URL string can take different formats. For the purposes of this demonstration, it takes the form `corbaloc:rir:/SimpleObject`. This form of the corbaloc URL string requires the use of a configuration variable, `initial_references:SimpleObject:reference`, in the configuration domain. When you submit the JCL in *orbixhlq*.DEMOS.CICS.PLI.BLD.JCL(UPDTCONF), it automatically adds this configuration entry to the configuration domain:

```
initial_references:SimpleObject:reference = "IOR…";
```

The IOR value is taken from the *orbixhlq*.DEMOS.IORS(SIMPLE) member.

See "STR2OBJ" on page 404 for more details of the various forms of corbaloc URL strings and the ways you can use them.

Step 7—Configure and run client adapter

The client adapter must now be configured before you can start the client as a CICS transaction. See the *CICS Adapters Administrator's Guide* for details of how to configure the client adapter.

When you have configured the client adapter, you can run it by submitting the following JCL:

```
orbixhlq.JCL(MFCLA)
```

Running the Demonstrations

Overview

This section provides a summary of what you need to do to successfully run the supplied demonstrations.

In this section

This section discusses the following topics:

Running Batch Client against CICS Server

Overview

This subsection describes what you need to do to successfully run the demonstration batch client against the demonstration CICS server. It also provides an overview of the output produced.

Steps

The steps to run the demonstration CICS server against the demonstration batch client are:

1. Ensure that all the steps in "Preparing the Server to Run in CICS" on page 118 have been successfully completed.
2. Run the batch client as described in "Running the Server and Client" on page 44.

CICS server output

The CICS server sends the following output to the CICS region:

```
Creating the simple_persistent object
Writing out the object reference
Giving control to the ORB to process Requests

Operation call_me() called
```

Batch client output

The batch client produces the following output:

```
simple_persistent demo
======================
Calling operation call_me…
Operation call_me completed (no results to display)

End of the simple_persistent demo
```

Running CICS Client against Batch Server

Overview

This subsection describes what you need to do to successfully run the demonstration CICS client against the demonstration batch server. It also provides an overview of the output produced.

Steps

The steps to run the demonstration CICS client against the demonstration batch server are:

1. Ensure that all the steps in "Preparing the Client to Run in CICS" on page 128 have been successfully completed.
2. Run the CICS client by entering the transaction name, `SMCL`, in the relevant CICS region.

CICS client output

The CICS client sends the following output to the CICS region:

```
simple_persistent demo
======================
Calling operation call_me…
Operation call_me completed (no results to display)

End of the simple_persistent demo
```

The CICS client sends the following output to the CICS terminal:

```
Simple transaction completed
```

Batch server output

The batch server produces the following output:

```
Creating the simple_persistent object
Writing out the object reference
Giving control to the ORB to proces Requests

Operation call_me() called
```

IDL Interfaces

The CORBA Interface Definition Language (IDL) is used to describe the interfaces of objects in an enterprise application. An object's interface describes that object to potential clients through its attributes and operations, and their signatures. This chapter describes IDL semantics and uses.

In this chapter

This chapter discusses the following topics:

IDL

Overview

An IDL-defined object can be implemented in any language that IDL maps to, including C++, Java, PL/I, and COBOL. By encapsulating object interfaces within a common language, IDL facilitates interaction between objects regardless of their actual implementation. Writing object interfaces in IDL is therefore central to achieving the CORBA goal of interoperability between different languages and platforms.

IDL standard mappings

CORBA defines standard mappings from IDL to several programming languages, including C++, Java, PL/I, and COBOL. Each IDL mapping specifies how an IDL interface corresponds to a language-specific implementation. The Orbix IDL compiler uses these mappings to convert IDL definitions to language-specific definitions that conform to the semantics of that language.

Overall structure

You create an application's IDL definitions within one or more IDL modules. Each module provides a naming context for the IDL definitions within it. Modules and interfaces form naming scopes, so identifiers defined inside an interface need to be unique only within that interface.

IDL definition structure

In the following example, two interfaces, `Bank` and `Account`, are defined within the `BankDemo` module:

```
module BankDemo
{
interface Bank {
        //…
    };

    interface Account {
        //…
    };
};
```

Modules and Name Scoping

Resolving a name

To resolve a name, the IDL compiler conducts a search among the following scopes, in the order outlined:

1. The current interface.
2. Base interfaces of the current interface (if any).
3. The scopes that enclose the current interface.

Referencing interfaces

Interfaces can reference each other by name alone within the same module. If an interface is referenced from outside its module, its name must be fully scoped with the following syntax:

`module-name::interface-name`

For example, the fully scoped names of the `Bank` and `Account` interfaces shown in "IDL definition structure" on page 136 are, respectively, `BankDemo::Bank` and `BankDemo::Account`.

Nesting restrictions

A module cannot be nested inside a module of the same name. Likewise, you cannot directly nest an interface inside a module of the same name. To avoid name ambiguity, you can provide an intervening name scope as follows:

```
module A
{
    module B
    {
        interface A {
            //…
        };
    };
};
```

Interfaces

In this section

The following topics are discussed in this section:

Overview

Interfaces are the fundamental abstraction mechanism of CORBA. An interface defines a type of object, including the operations that object supports in a distributed enterprise application.

Every CORBA object has exactly one interface. However, the same interface can be shared by many CORBA objects in a system. CORBA object references specify CORBA objects (that is, interface instances). Each reference denotes exactly one object, which provides the only means by which that object can be accessed for operation invocations.

Because an interface does not expose an object's implementation, all members are public. A client can access variables in an object's implementation only through an interface's operations and attributes.

Operations and attributes

An IDL interface generally defines an object's behavior through operations and attributes:

- Operations of an interface give clients access to an object's behavior. When a client invokes an operation on an object, it sends a message to that object. The ORB transparently dispatches the call to the object, whether it is in the same address space as the client, in another address space on the same machine, or in an address space on a remote machine.

- An IDL attribute is short-hand for a pair of operations that get and, optionally, set values in an object.

Account interface IDL sample

In the following example, the `Account` interface in the `BankDemo` module describes the objects that implement the bank accounts:

```
module BankDemo
{
    typedef float CashAmount; // Type for representing cash
    typedef string AccountId; //Type for representing account ids
    //…
    interface Account {
        readonly attribute AccountId  account_id;
        readonly attribute CashAmount balance;

        void
        withdraw(in CashAmount amount)
        raises (InsufficientFunds);

        void
        deposit(in CashAmount amount);
    };
};
```

Code explanation

This interface has two readonly attributes, `AccountId` and `balance`, which are respectively defined as typedefs of the `string` and `float` types. The interface also defines two operations, `withdraw()` and `deposit()`, which a client can invoke on this object.

Interface Contents

IDL interface components

An IDL interface definition typically has the following components.

- Operation definitions.
- Attribute definitions
- Exception definitions.
- Type definitions.
- Constant definitions.

Of these, operations and attributes must be defined within the scope of an interface, all other components can be defined at a higher scope.

Operations

Overview

Operations of an interface give clients access to an object's behavior. When a client invokes an operation on an object, it sends a message to that object. The ORB transparently dispatches the call to the object, whether it is in the same address space as the client, in another address space on the same machine, or in an address space on a remote machine.

Operation components

IDL operations define the signature of an object's function, which client invocations on that object must use. The signature of an IDL operation is generally composed of three components:

- Return value data type.
- Parameters and their direction.
- Exception clause.

An operation's return value and parameters can use any data types that IDL supports.

Note: Not all CORBA 2.3 IDL data types are supported by PL/I or COBOL.

Operations IDL sample

In the following example, the `Account` interface defines two operations, `withdraw()` and `deposit()`, and an `InsufficientFunds` exception:

```
module BankDemo
{
    typedef float CashAmount;  // Type for representing cash
    //...
    interface Account {
        exception InsufficientFunds {};

        void
        withdraw(in CashAmount amount)
        raises (InsufficientFunds);

        void
        deposit(in CashAmount amount);
    };
};
```

Code explanation

On each invocation, both operations expect the client to supply an argument for the `amount` parameter, and return `void`. Invocations on the `withdraw()` operation can also raise the `InsufficientFunds` exception, if necessary.

Parameter direction

Each parameter specifies the direction in which its arguments are passed between client and object. Parameter-passing modes clarify operation definitions and allow the IDL compiler to accurately map operations to a target programming language. The PL/I runtime uses parameter-passing modes to determine in which direction or directions it must marshal a parameter.

Parameter-passing mode qualifiers

There are three parameter-passing mode qualifiers:

`in`	This means that the parameter is initialized only by the client and is passed to the object.
`out`	This means that the parameter is initialized only by the object and returned to the client.
`inout`	This means that the parameter is initialized by the client and passed to the server; the server can modify the value before returning it to the client.

In general, you should avoid using `inout` parameters. Because an `inout` parameter automatically overwrites its initial value with a new value, its usage assumes that the caller has no use for the parameter's original value. Thus, the caller must make a copy of the parameter in order to retain that value. By using the two parameters, `in` and `out`, the caller can decide for itself when to discard the parameter.

One-way operations

By default, IDL operations calls are *synchronous*—that is, a client invokes an operation on an object and blocks until the invoked operation returns. If an operation definition begins with the keyword, `oneway`, a client that calls the operation remains unblocked while the object processes the call.

Note: The PL/I runtime does not support one-way operations.

Attributes

Attributes overview

An interface's attributes correspond to the variables that an object implements. Attributes indicate which variable in an object are accessible to clients.

Qualified and unqualified attributes

Unqualified attributes map to a pair of `get` and `set` functions in the implementation language, which allow client applications to read and write attribute values. An attribute that is qualified with the `readonly` keyword maps only to a `get` function.

IDL readonly attributes sample

For example the `Account` interface defines two readonly attributes, `AccountId` and `balance`. These attributes represent information about the account that only the object's implementation can set; clients are limited to readonly access:

```
module BankDemo
{
    typedef float CashAmount; // Type for representing cash
    typedef string AccountId; //Type for representing account ids
    //…
    interface Account {
        readonly attribute AccountId  account_id;
        readonly attribute CashAmount balance;

        void
        withdraw(in CashAmount amount)
        raises (InsufficientFunds);

        void
        deposit(in CashAmount amount);
    };
};
```

Code explanation

The `Account` interface has two readonly attributes, `AccountId` and `balance`, which are respectively defined as typedefs of the `string` and `float` types. The interface also defines two operations, `withdraw()` and `deposit()`, which a client can invoke on this object.

Exceptions

IDL and exceptions

IDL operations can raise one or more CORBA-defined system exceptions. You can also define your own exceptions and explicitly specify these in an IDL operation. An IDL exception is a data structure that can contain one or more member fields, formatted as follows:

```
exception exception-name {
    [member;]…
};
```

Exceptions that are defined at module scope are accessible to all operations within that module; exceptions that are defined at interface scope are accessible on to operations within that interface.

The raises clause

After you define an exception, you can specify it through a `raises` clause in any operation that is defined within the same scope. A `raises` clause can contain multiple comma-delimited exceptions:

```
return-val operation-name( [params-list] )
    raises( exception-name[, exception-name] );
```

Example of IDL-defined exceptions

The `Account` interface defines the `InsufficientFunds` exception with a single member of the `string` data type. This exception is available to any operation within the interface. The following IDL defines the `withdraw()` operation to raise this exception when the withdrawal fails:

```
module BankDemo
{
    typedef float CashAmount;  // Type for representing cash
    //…
    interface Account {
        exception InsufficientFunds {};

        void
        withdraw(in CashAmount amount)
        raises (InsufficientFunds);
        //…
    };
};
```

Empty Interfaces

Defining empty interfaces

IDL allows you to define empty interfaces. This can be useful when you wish to model an abstract base interface that ties together a number of concrete derived interfaces.

IDL empty interface sample

In the following example, the CORBA `PortableServer` module defines the abstract `Servant Manager` interface, which serves to join the interfaces for two servant manager types, `ServantActivator` and `ServantLocator`:

```
module PortableServer
{
    interface ServantManager {};

    interface ServantActivator : ServantManager {
        //…
    };

    interface ServantLocator : ServantManager {
        //…
    };
};
```

Inheritance of Interfaces

Inheritance overview

An IDL interface can inherit from one or more interfaces. All elements of an inherited, or *base* interface, are available to the *derived* interface. An interface specifies the base interfaces from which it inherits, as follows:

```
interface new-interface : base-interface[, base-interface]…
{…};
```

Inheritance interface IDL sample

In the following example, the `CheckingAccount` and `SavingsAccount` interfaces inherit from the `Account` interface, and implicitly include all its elements:

```
module BankDemo{
    typedef float CashAmount;  // Type for representing cash
    interface Account {
        //…
    };

    interface CheckingAccount : Account {
        readonly attribute CashAmount overdraftLimit;
        boolean orderCheckBook ();
    };

    interface SavingsAccount : Account {
        float calculateInterest ();
    };
};
```

Code sample explanation

An object that implements the `CheckingAccount` interface can accept invocations on any of its own attributes and operations as well as invocations on any of the elements of the `Account` interface. However, the actual implementation of elements in a `CheckingAccount` object can differ from the implementation of corresponding elements in an `Account` object. IDL inheritance only ensures type-compatibility of operations and attributes between base and derived interfaces.

Multiple Inheritance

Multiple inheritance IDL sample

In the following IDL definition, the `BankDemo` module is expanded to include the `PremiumAccount` interface, which inherits from the `CheckingAccount` and `SavingsAccount` interfaces:

```
module BankDemo {
    interface Account {
        //…
    };

    interface CheckingAccount : Account {
        //…
    };

    interface SavingsAccount : Account {
        //…
    };

    interface PremiumAccount :
        CheckingAccount, SavingsAccount {
        //…
    };
};
```

Multiple inheritance constraints

Multiple inheritance can lead to name ambiguity among elements in the base interfaces. The following constraints apply:

- Names of operations and attributes must be unique across all base interfaces.
- If the base interfaces define constants, types, or exceptions of the same name, references to those elements must be fully scoped.

Inheritance hierarchy diagram

Figure 4 shows the inheritance hierarchy for the `Account` interface, which is defined in “Multiple inheritance IDL sample” on page 147.

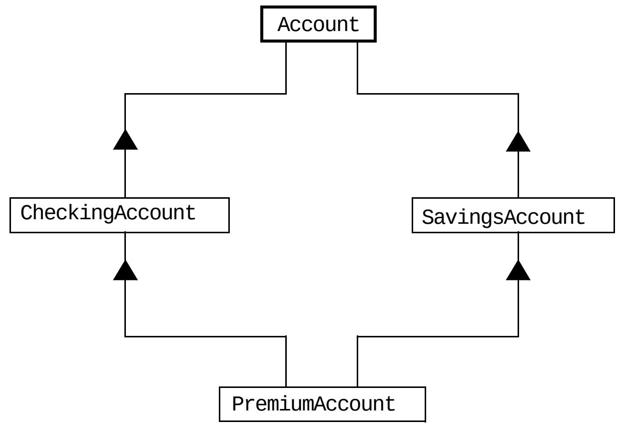

Figure 4: *Inheritance Hierarchy for PremiumAccount Interface*

Inheritance of the Object Interface

User-defined interfaces

All user-defined interfaces implicitly inherit the predefined interface `Object`. Thus, all `Object` operations can be invoked on any user-defined interface. You can also use `Object` as an attribute or parameter type to indicate that any interface type is valid for the attribute or parameter.

Object locator IDL sample

For example, the following operation `getAnyObject()` serves as an all-purpose object locator:

```
interface ObjectLocator {
    void getAnyObject (out Object obj);
};
```

Note: It is illegal in IDL syntax to explicitly inherit the `Object` interface.

Inheritance Redefinition

Overview

A derived interface can modify the definitions of constants, types, and exceptions that it inherits from a base interface. All other components that are inherited from a base interface cannot be changed.

Inheritance redefinition IDL sample

In the following example, the `CheckingAccount` interface modifies the definition of the `InsufficientFunds` exception, which it inherits from the `Account` interface:

```
module BankDemo
{
    typedef float CashAmount;  // Type for representing cash
    //…
    interface Account {
        exception InsufficientFunds {};
        //…
    };
    interface CheckingAccount : Account {
        exception InsufficientFunds {
            CashAmount overdraftLimit;
        };
    };
    //…
};
```

Note: While a derived interface definition cannot override base operations or attributes, operation overloading is permitted in interface implementations for those languages, such as C++, which support it. However, PL/I does not support operation overloading.

Forward Declaration of IDL Interfaces

Overview

An IDL interface must be declared before another interface can reference it. If two interfaces reference each other, the module must contain a forward declaration for one of them; otherwise, the IDL compiler reports an error. A forward declaration only declares the interface's name; the interface's actual definition is deferred until later in the module.

Forward declaration IDL sample

In the following example, the `Bank` interface defines a `create_account()` and `find_account()` operation, both of which return references to `Account` objects. Because the `Bank` interface precedes the definition of the `Account` interface, `Account` is forward-declared:

```
module BankDemo
{
    typedef float CashAmount; // Type for representing cash
    typedef string AccountId; //Type for representing account ids

    // Forward declaration of Account
    interface Account;

    // Bank interface...used to create Accounts
    interface Bank {
        exception AccountAlreadyExists { AccountId account_id; };
        exception AccountNotFound      { AccountId account_id; };

        Account
        find_account(in AccountId account_id)
        raises(AccountNotFound);

        Account
        create_account(
            in AccountId account_id,
            in CashAmount initial_balance
        ) raises (AccountAlreadyExists);
    };

    // Account interface…used to deposit, withdraw, and query
    // available funds.
    interface Account { //…
    };
};
```

Local Interfaces

Overview

An interface declaration that contains the IDL `local` keyword defines a *local interface*. An interface declaration that omits this keyword can be referred to as an *unconstrained interface*, to distinguish it from local interfaces. An object that implements a local interface is a *local object*.

Note: The PL/I runtime and the Orbix IDL compiler backend for PL/I do not support local interfaces.

Valuetypes

Overview

Valuetypes enable programs to pass objects by value across a distributed system. This type is especially useful for encapsulating lightweight data such as linked lists, graphs, and dates.

Note: The PL/I runtime and the Orbix IDL compiler backend for PL/I do not support valuetypes.

Abstract Interfaces

Overview

An application can use abstract interfaces to determine at runtime whether an object is passed by reference or by value.

Note: The PL/I runtime and the Orbix IDL compiler backend for PL/I do not support abstract interfaces.

IDL Data Types

In this section

The following topics are discussed in this section:

Data type categories

In addition to IDL module, interface, valuetype, and exception types, IDL data types can be grouped into the following categories:

- Built-in types such as `short`, `long`, and `float`.
- Extended built-in types such as `long long` and `wstring`.
- Complex types such as `enum`, `struct`, and `string`.
- Pseudo objects.

Note: Not all CORBA 2.3 IDL data types are supported by PL/I or COBOL.

Built-in Data Types

List of types, sizes, and values

Table 18 shows a list of CORBA IDL built-in data types (where the ≤ symbol means ’less than or equal to’).

Table 18: *Built-in IDL Data Types, Sizes, and Values*

Data type	Size	Range of values
short	≤ 16 bits	$-2^{15} \ldots 2^{15}-1$
unsigned short[a]	≤ 16 bits	$0 \ldots 2^{16}-1$
long	≤ 32 bits	$-2^{31} \ldots 2^{31}-1$
unsigned long[b]	≤ 32 bits	$0 \ldots 2^{32}-1$
float	≤ 32 bits	IEEE single-precision floating point numbers
double	≤ 64 bits	IEEE double-precision floating point numbers
char	≤ 8 bits	ISO Latin-1
string	Variable length	ISO Latin-1, except NUL
string<bound>[c]	Variable length	ISO Latin-1, except NUL
boolean	Unspecified	TRUE or FALSE
octet	≤ 8 bits	`0x0` to `0xff`
any	Variable length	Universal container type

a. The PL/I range for the `unsigned short` type is restricted to $0 \ldots 2^{15}-1$.

b. The PL/I range for the unsigned long type is restricted to $0 \ldots 2^{31}-1$

c. The PL/I range for a bounded string is restricted to a range of 1–32767 characters.

Integer types

With the exception of unsigned short, unsigned long , and bounded string types, the full IDL range of values of each of the types listed in Table 18 can be marshaled to and from the PL/I runtime. Due to a limitation of the PL/I compiler for MVS & VM, the upper range of values for `unsigned short` and `unsigned long` types are the same as those for `short` and `long` types.

Floating point types

The float and double types follow IEEE specifications for single-precision and double-precision floating point values, and on most platforms map to native IEEE floating point types.

Char type

The `char` type can hold any value from the ISO Latin-1 character set. Code positions 0-127 are identical to ASCII. Code positions 128-255 are reserved for special characters in various European languages, such as accented vowels.

String type

The `string` type can hold any character from the ISO Latin-1 character set, except `NUL`. IDL prohibits embedded `NUL` characters in strings. Unbounded string lengths are generally constrained only by memory limitations. A bounded string, such as `string<10>`, can hold only the number of characters specified by the bounds, excluding the terminating `NUL` character. Thus, a `string<6>` can contain the six-character string, cheese.

Bounded and unbounded strings

The declaration statement can optionally specify the string's maximum length, thereby determining whether the string is bounded or unbounded:

```
string[length] name
```

For example, the following code declares the `ShortString` type, which is a bounded string with a maximum length of 10 characters:

```
typedef string<10> ShortString;
attribute ShortString shortName; // max length is 10 chars
```

Due to the limitations in PL/I, a bounded string can have a maximum length of 32767 characters.

Octet type

`Octet` types are guaranteed not to undergo any conversions in transit. This lets you safely transmit binary data between different address spaces. Avoid using the `char` type for binary data, inasmuch as characters might be subject to translation during transmission. For example, if a client that uses ASCII sends a string to a server that uses EBCDIC, the sender and receiver are liable to have different binary values for the string's characters.

Any type

The any type allows specification of values that express any IDL type, which is determined at runtime; thereby allowing a program to handle values whose types are not known at compile time. An any logically contains a `TypeCode` and a value that is described by the `TypeCode`. A client or server can construct an any to contain an arbitrary type of value and then pass this call in a call to the operation. A process receiving an any must determine what type of value it stores and then extract the value via the TypeCode. See the *CORBA Programmer's Guide, C++* for more details about the any type.

Extended Built-in Data Types

List of types, sizes, and values

Table 19 shows a list of CORBA IDL extended built-in data types (where the ≤ symbol means 'less than or equal to').

Table 19: *Extended built-in IDL Data Types, Sizes, and Values*

Data Type	Size	Range of Values
long long[a]	≤ 64 bits	$-2^{63}...2^{63}-1$
unsigned long long[a]	≤ 64 bits	$0...-2^{64}-1$
long double[b]	≤ 79 bits	IEEE double-extended floating point number, with an exponent of at least 15 bits in length and signed fraction of at least 64 bits. `long double` type is currently not supported on Windows NT.
wchar	Unspecified	Arbitrary codesets
wstring	Variable length	Arbitrary codesets
fixed[c]	Unspecified	≤ 31significant digits

a. Due to compiler restrictions, the PL/I range of values for the `long long` and `unsigned long long` types is the same range as for a `long` type (that is, $0...2^{31}-1$).

b. Due to compiler restrictions, the PL/I range of values for the `long double` type is the same range as for a double type (that is, ≤ 64 bits).

c. Due to compiler restrictions, the PL/I range of values for the fixed type is ≤ 15 significant digits.

Long long type

The 64-bit integer types, `long long` and unsigned `long long`, support numbers that are too large for 32-bit integers. Platform support varies. If you compile IDL that contains one of these types on a platform that does not support it, the compiler issues an error.

Long double type

Like 64-bit integer types, platform support varies for the `long double` type, so usage can yield IDL compiler errors.

Wchar type

The `wchar` type encodes wide characters from any character set. The size of a `wchar` is platform-dependent. Because Orbix currently does not support character set negotiation, use this type only for applications that are distributed across the same platform.

Wstring type

The `wstring` type is the wide-character equivalent of the `string` type. Like `string` types, `wstring` types can be unbounded or bounded. Wide strings can contain any character except `NUL`.

Fixed type

IDL specifies that the `fixed` type provides fixed-point arithmetic values with up to 31 significant digits. However, due to restrictions in the PL/I compiler for MVS & VM, only up to 15 significant digits are supported.

You specify a `fixed` type with the following format:

```
typedef fixed<digit-size,scale> name
```

The format for the fixed type can be explained as follows:

- The *`digit-size`* represents the number's length in digits. The maximum value for *`digit-size`* is 31 and it must be greater than *`scale`*. A `fixed` type can hold any value up to the maximum value of a `double` type.
- If *`scale`* is a positive integer, it specifies where to place the decimal point relative to the rightmost digit. For example, the following code declares a fixed type, `CashAmount`, to have a digit size of 10 and a scale of 2:

  ```
  typedef fixed<10,2> CashAmount;
  ```

  Given this typedef, any variable of the `CashAmount` type can contain values of up to (+/-)99999999.99.

- If *scale* is a negative integer, the decimal point moves to the right by the number of digits specified for *scale*, thereby adding trailing zeros to the fixed data type's value. For example, the following code declares a fixed type, `bigNum`, to have a digit size of `3` and a scale of `-4`:

  ```
  typedef fixed <3,-4> bigNum;
  bigNum myBigNum;
  ```

  If `myBigNum` has a value of `123`, its numeric value resolves to `1230000`. Definitions of this sort allow you to efficiently store numbers with trailing zeros.

Constant fixed types

Constant fixed types can also be declared in IDL, where *digit-size* and *scale* are automatically calculated from the constant value. For example:

```
module Circle {
    const fixed pi = 3.142857;
};
```

This yields a fixed type with a digit size of `7`, and a scale of `6`.

Fixed type and decimal fractions

Unlike IEEE floating-point values, the `fixed` type is not subject to representational errors. IEEE floating point values are liable to inaccurately represent decimal fractions unless the value is a fractional power of 2. For example, the decimal value 0.1 cannot be represented exactly in IEEE format. Over a series of computations with floating-point values, the cumulative effect of this imprecision can eventually yield inaccurate results.

The `fixed` type is especially useful in calculations that cannot tolerate any imprecision, such as computations of monetary values.

Complex Data Types

IDL complex data types

IDL provide the following complex data types:

- Enums.
- Structs.
- Multi-dimensional fixed-sized arrays.
- Sequences.

Enum Data Type

Overview

An enum (enumerated) type lets you assign identifiers to the members of a set of values.

Enum IDL sample

For example, you can modify the `BankDemo` IDL with the `balanceCurrency` enum type:

```
module BankDemo {
    enum Currency {pound, dollar, yen, franc};

    interface Account {
        readonly attribute CashAmount balance;
        readonly attribute Currency balanceCurrency;
        //…
    };
};
```

In the preceding example, the `balanceCurrency` attribute in the `Account` interface can take any one of the values `pound, dollar, yen`, or `franc`.

Ordinal values of enum type

The ordinal values of an enum type vary according to the language implementation. The CORBA specification only guarantees that the ordinal values of enumerated types monotonically increase from left to right. Thus, in the previous example, `dollar` is greater than `pound, yen` is greater than `dollar`, and so on. All enumerators are mapped to a 32-bit type.

Struct Data Type

Overview

A struct type lets you package a set of named members of various types.

Struct IDL sample

In the following example, the `CustomerDetails` struct has several members. The `getCustomerDetails()` operation returns a struct of the `CustomerDetails` type, which contains customer data:

```
module BankDemo{
        struct CustomerDetails {
            string custID;
            string lname;
            string fname;
            short age;
            //…
        };

        interface Bank {
            CustomerDetails getCustomerDetails
                (in string custID);
            //…
        };
};
```

Note: A struct type must include at least one member. Because a struct provides a naming scope, member names must be unique only within the enclosing structure.

Union Data Type

Overview

A union type lets you define a structure that can contain only one of several alternative members at any given time. A union type saves space in memory, because the amount of storage required for a union is the amount necessary to store its largest member.

Union declaration syntax

You declare a union type with the following syntax:

```
union name switch (discriminator) {
    case label1 : element-spec;
    case label2 : element-spec;
    […]
    case labeln : element-spec;
    [default : element-spec;]
};
```

Discriminated unions

All IDL unions are *discriminated*. A discriminated union associates a constant expression (`label1…labeln`) with each member. The discriminator's value determines which of the members is active and stores the union's value.

IDL union date sample

The following IDL defines a `Date` union type, which is discriminated by an enum value:

```
enum dateStorage
{ numeric, strMMDDYY, strDDMMYY };

struct DateStructure {
    short Day;
    short Month;
    short Year;
};

union Date switch (dateStorage) {
    case numeric: long digitalFormat;
    case strMMDDYY:
    case strDDMMYY: string stringFormat;
    default: DateStructure structFormat;
};
```

Sample explanation

Given the preceding IDL:

- If the discriminator value for `Date` is numeric, the `digitalFormat` member is active.
- If the discriminator's value is `strMMDDYY` or `strDDMMYY`, the `stringFormat` member is active.
- If neither of the preceding two conditions apply, the default `structFormat` member is active.

Rules for union types

The following rules apply to union types:

- A union's discriminator can be `integer, char, boolean` or `enum`, or an alias of one of these types; all `case` label expressions must be compatible with the relevant type.
- Because a union provides a naming scope, member names must be unique only within the enclosing union.
- Each union contains a pair of values: the discriminator value and the active member.
- IDL unions allow multiple case labels for a single member. In the previous example, the `stringFormat` member is active when the discriminator is either `strMMDDYY` or `strDDMMYY`.
- IDL unions can optionally contain a `default` case label. The corresponding member is active if the discriminator value does not correspond to any other label.

Arrays

Overview

IDL supports multi-dimensional fixed-size arrays of any IDL data type, with the following syntax (where *dimension-spec* must be a non-zero positive constant integer expression):

```
[typedef] element-type array-name [dimension-spec]…
```

IDL does not allow open arrays. However, you can achieve equivalent functionality with sequence types.

Array IDL sample

For example, the following piece of code defines a two-dimensional array of bank accounts within a portfolio:

```
typedef Account portfolio[MAX_ACCT_TYPES][MAX_ACCTS]
```

Note: For an array to be used as a parameter, an attribute, or a return value, the array must be named by a typedef declaration. You can omit a typedef declaration only for an array that is declared within a structure definition.

Array indexes

Because of differences between implementation languages, IDL does not specify the origin at which arrays are indexed. For example, C and C++ array indexes always start at 0, while PL/I, COBOL, and Pascal use an origin of 1. Consequently, clients and servers cannot exchange array indexes unless they both agree on the origin of array indexes and make adjustments as appropriate for their respective implementation languages. Usually, it is easier to exchange the array element itself instead of its index.

Sequence

Overview

IDL supports sequences of any IDL data type with the following syntax:

`[typedef] sequence < `*`element-type[, max-elements]`*` > `*`sequence-name`*

An IDL sequence is similar to a one-dimensional array of elements; however, its length varies according to its actual number of elements, so it uses memory more efficiently.

For a sequence to be used as a parameter, an attribute, or a return value, the sequence must be named by a typedef declaration, to be used as a parameter, an attribute, or a return value. You can omit a typedef declaration only for a sequence that is declared within a structure definition.

A sequence's element type can be of any type, including another sequence type. This feature is often used to model trees.

Bounded and unbounded sequences

The maximum length of a sequence can be fixed (bounded) or unfixed (unbounded):

- Unbounded sequences can hold any number of elements, up to the memory limits of your platform.
- Bounded sequences can hold any number of elements, up to the limit specified by the bound.

Bounded and unbounded IDL definitions

The following code shows how to declare bounded and unbounded sequences as members of an IDL struct:

```
struct LimitedAccounts {
    string bankSortCode<10>;
    sequence<Account, 50> accounts; // max sequence length is 50
};

struct UnlimitedAccounts {
    string bankSortCode<10>;
    sequence<Account> accounts; // no max sequence length
};
```

Pseudo Object Types

Overview

CORBA defines a set of pseudo-object types that ORB implementations use when mapping IDL to a programming language. These object types have interfaces defined in IDL; however, these object types do not have to follow the normal IDL mapping rules for interfaces and they are not generally available in your IDL specifications.

Note: The PL/I runtime and the Orbix IDL compiler backend for PL/I do not support all pseudo object types.

Defining Data Types

In this section

This section contains the following subsections:

Using typedef

With `typedef`, you can define more meaningful or simpler names for existing data types, regardless of whether those types are IDL-defined or user-defined.

Typedef identifier IDL sample

The following code defines the `typedef` identifier, `StandardAccount`, so that it can act as an alias for the `Account` type in later IDL definitions:

```
module BankDemo {
    interface Account {
        //…
    };

    typedef Account StandardAccount;
};
```

Constants

Overview

IDL lets you define constants of all built-in types except the any type. To define a constant's value, you can use either another constant (or constant expression) or a literal. You can use a constant wherever a literal is permitted.

Integer constants

IDL accepts integer literals in decimal, octal, or hexadecimal:

```
const short     I1 = -99;
const long      I2 = 0123;  // Octal 123, decimal 83
const long long I3 = 0x123; // Hexadecimal 123, decimal 291
const long long I4 = +0xaB; // Hexadecimal ab, decimal 171
```

Both unary plus and unary minus are legal.

Floating-point constants

Floating-point literals use the same syntax as C++:

```
const float       f1 = 3.1e-9; // Integer part, fraction part,
                                // exponent
const double      f2 = -3.14;  // Integer part and fraction part
const long double f3 = .1      // Fraction part only
const double      f4 = 1.      // Integer part only
const double      f5 = .1E12   // Fraction part and exponent
const double      f6 = 2E12    // Integer part and exponent
```

Character and string constants

Character constants use the same escape sequences as C++:

Example 7: *List of character constants (Sheet 1 of 2)*

```
const char C1 = 'c';        // the character c
const char C2 = '\007';     // ASCII BEL, octal escape
const char C3 = '\x41';     // ASCII A, hex escape
const char C4 = '\n';       // newline
const char C5 = '\t';       // tab
const char C6 = '\v';       // vertical tab
const char C7 = '\b';       // backspace
const char C8 = '\r';       // carriage return
const char C9 = '\f';       // form feed
const char C10 = '\a';      // alert
```

Example 7: *List of character constants (Sheet 2 of 2)*

```
const char C11 = '\\';      // backslash
const char C12 = '\?';      // question mark
const char C13 = '\'';      // single quote
// String constants support the same escape sequences as C++
const string S1 = "Quote: \"";      // string with double quote
const string S2 = "hello world";    // simple string
const string S3 = "hello" " world"; // concatenate
const string S4 = "\xA" "B";        // two characters
                                    // ('\xA' and 'B'),
                               // not the single character '\xAB'
```

Wide character and string constants

Wide character and string constants use C++ syntax. Use universal character codes to represent arbitrary characters. For example:

```
const wchar     C = L'X';
const wstring   GREETING = L"Hello";
const wchar     OMEGA = L'\u03a9';
const wstring   OMEGA_STR = L"Omega: \u3A9";
```

IDL files always use the ISO Latin-1 code set; they cannot use Unicode or other extended character sets.

Boolean constants

Boolean constants use the `FALSE` and `TRUE` keywords. Their use is unnecessary, inasmuch as they create unnecessary aliases:

```
// There is no need to define boolean constants:
const CONTRADICTION = FALSE;    // Pointless and confusing
const TAUTOLOGY = TRUE;         // Pointless and confusing
```

Octet constants

Octet constants are positive integers in the range 0-255.

```
const octet O1 = 23;
const octet O2 = 0xf0;
```

Octet constants were added with CORBA 2.3; therefore, ORBs that are not compliant with this specification might not support them.

Fixed-point constants

For fixed-point constants, you do not explicitly specify the digits and scale. Instead, they are inferred from the initializer. The initializer must end in `d` or `D`. For example:

```
// Fixed point constants take digits and scale from the
// initializer:
const fixed val1 = 3D;          // fixed<1,0>
const fixed val2 = 03.14d;      // fixed<3,2>
const fixed val3 = -03000.00D;  // fixed<4,0>
const fixed val4 = 0.03D;       // fixed<3,2>
```

The type of a fixed-point constant is determined after removing leading and trailing zeros. The remaining digits are counted to determine the digits and scale. The decimal point is optional.

Currently, there is no way to control the scale of a constant if it ends in trailing zeros.

Enumeration constants

Enumeration constants must be initialized with the scoped or unscoped name of an enumerator that is a member of the type of the enumeration. For example:

```
enum Size { small, medium, large }

const Size DFL_SIZE = medium;
const Size MAX_SIZE = ::large;
```

Enumeration constants were added with CORBA 2.3; therefore, ORBs that are not compliant with this specification might not support them.

Constant Expressions

Overview

IDL provides a number of arithmetic and bitwise operators. The arithmetic operators have the usual meaning and apply to integral, floating-point, and fixed-point types (except for %, which requires integral operands). However, these operators do not support mixed-mode arithmetic: you cannot, for example, add an integral value to a floating-point value.

Arithmetic operators

The following code contains several examples of arithmetic operators:

```
// You can use arithmetic expressions to define constants.
const long MIN = -10;
const long MAX = 30;
const long DFLT = (MIN + MAX) / 2;

// Can't use 2 here
const double TWICE_PI = 3.1415926 * 2.0;

// 5% discount
const fixed DISCOUNT = 0.05D;
const fixed PRICE = 99.99D;

// Can't use 1 here
const fixed NET_PRICE = PRICE * (1.0D - DISCOUNT);
```

Evaluating expressions for arithmetic operators

Expressions are evaluated using the type promotion rules of C++. The result is coerced back into the target type. The behavior for overflow is undefined, so do not rely on it. Fixed-point expressions are evaluated internally with 31 bits of precision, and results are truncated to 15 digits.

Bitwise operators

Bitwise operators only apply to integral types. The right-hand operand must be in the range 0-63. The right-shift operator, >>, is guaranteed to insert zeros on the left, regardless of whether the left-hand operand is signed or unsigned.

```
// You can use bitwise operators to define constants.
const long ALL_ONES = -1;               // 0xffffffff
const long LHW_MASK = ALL_ONES << 16;   // 0xffff0000
const long RHW_MASK = ALL_ONES >> 16;   // 0x0000ffff
```

IDL guarantees two's complement binary representation of values.

Precedence

The precedence for operators follows the rules for C++. You can override the default precedence by adding parentheses.

IDL-to-PL/I Mapping

The CORBA Interface Definition Language (IDL) is used to define interfaces that are offered by servers on your network. This chapter describes how the Orbix IDL compiler maps IDL data types to PL/I. It shows, with examples, how each IDL type is represented in PL/I.

In this chapter

This chapter discusses the following topics:

Note the following points:

- For the purposes of the examples shown in this chapter, the member name for each example is the same as the interface name, unless otherwise stated.
- For the purposes of PL/I application development, Orbix closely follows the IDL-to-PL/I mapping rules described in the OMG specification. To provide compatibility for both PL/I compilers that Orbix supports, Orbix generally only differs from these rules where the PL/I compiler for MVS & VM does not support a particular feature, such as `UNSIGNED FIXED BIN(32)`. See `www.omg.org` for details about the IDL-to-PL/I mapping specification.
- See "IDL Interfaces" on page 135 for more details of the IDL types discussed in this chapter.

Mapping for Identifier Names

Overview

This section describes how IDL identifier names are mapped to PL/I.

Standard mapping rule

The Orbix IDL compiler uses the following basic rule to generate PL/I identifiers unless you use the `-O` argument to generate an alternative naming scheme (see “-O Argument” on page 254 for more details):

```
moduleName_interfaceName_IDLvariableName
```

Further guidelines

The naming scheme for PL/I identifiers also adheres to the following guidelines:

- If the identifier is within a nested module, these module names are prefixed to the *`moduleName_interfaceName_IDLvariableName`* format.
- An identifier name that exceeds 31 characters is abbreviated to its first 26 characters, and is appended with an underscore followed by a four-character hash suffix.
- If an identifier name exceeds 31 characters and is a particular type that already ends with a particular suffix (for example, an argument block always ends in `_args`), the identifier name is abbreviated to its first 21 characters, and is appended with an underscore followed by a four-character hash suffix followed by its existing suffix. See “Mapping Very Long and Leading Underscored Names” on page 181.
- Upper case characters map to upper case, and lower case characters map to lower case. For example, `myName` in IDL maps to `myName` in PL/I.
- If the identifier is a PL/I keyword, the identifier is mapped with an `idl_` prefix. The Orbix IDL compiler supports the PL/I-reserved words pertaining to the IBM PL/I for MVS & VM V1R1M1 and Enterprise PL/I compilers.
- The first and last lines of a procedure are always capitalized, except for server implementation sub-procedures, which have a `proc_` prefix.

- If you specify the `-Mprocess` option, the mappings specified for mapping *`modulename/interfacename`* are used instead. See "Orbix IDL Compiler" on page 233 for more details.
- Identifiers defined at IDL file level, outside any modules or interfaces, have the IDL member name incorporated in their name. See "Example" on page 184 to see how such identifiers are mapped.

Mapping Very Long and Leading Underscored Names

Overview

This section describes how very long IDL identifier names, or identifiers within a module with a very long name, are mapped to PL/I.

Standard mapping rule

As stated in "Further guidelines" on page 179, if the identifier name exceeds 31 characters, and it is of a particular type that already ends with a particular suffix (for example, an argument block always ends in `_args`), this suffix is included in the generated name. In this case, the identifier name is abbreviated to its first 21 characters, and is appended with an underscore followed by a four-character hash suffix followed by the existing suffix.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
module BankLoans {
    interface Mortgages {
        float calculateMonthlyRepay(
            in long amountBorrowed,
            in float interestRate,
            in short durationBorrowedFor);
    };

    const float _special_rate=4.5;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the operation structure name for `calculateMonthlyRepay` as follows:

```
dcl 1 BankLoans_Mortgages_c_ee9c_args aligned like
      BankLoans_Mortgages_c_ee9c_type;
```

Avoiding the standard rule

You can use the `-O` argument with the Orbix IDL compiler, to avoid the standard way in which identifier names are abbreviated. You can do this by using the `-O` argument to set up an alternative mapping entry in the mapping member. For example, consider the following mapping member entry:

```
BankLoans/Mortgages/calculateMonthlyRepay calculateMonthlyRepay
```

Based on the preceding mapping member entry, the Orbix IDL compiler generates the operation structure name for `calculateMonthlyRepay` as follows:

```
dcl 1 calculateMonthlyRepay_args aligned like
      calculateMonthlyRepay_type;
```

The mapping for the `_special_rate` constant is as follows (in this case, the Orbix IDL compiler removes the leading underscore from the mapped PL/I name by default):

```
dcl 1 BankLoans_consts,
      3 special_rate          float dec(6)  init(4.5e+00);
```

Mapping for Basic Types

Overview

This section describes how basic IDL types are mapped to PL/I.

IDL-to-PL/I mapping for basic types

Table 20 shows the mapping rules for basic IDL types. The CORBA typedef name is provided for reference purposes only; the PL/I representation is used directly.

Table 20: *Mapping for Basic IDL Types*

IDL Type	CORBA Typedef Name	PL/I Representation
short	CORBA-short	FIXED BIN(15)
long	CORBA-long	FIXED BIN(31)
unsigned short	CORBA-unsigned-short	FIXED BIN(15)[a]
unsigned long	CORBA-unsigned-long	FIXED BIN(31)[a]
float	CORBA-float	FLOAT DEC(6)
double	CORBA-double	FLOAT DEC(16)
char	CORBA-char	CHAR(1)
boolean	CORBA-boolean	CHAR(1)
octet	CORBA-octet	CHAR(1)
enum	CORBA-enum	FIXED BIN(31)[a,b]
fixed<d,s>	Fixed<d,s>	FIXED DEC(d,s)
any	CORBA-any	See "Mapping for the Any Type" on page 204.
long long	CORBA-long-long	FIXED BIN(31)[b]

Table 20: *Mapping for Basic IDL Types*

IDL Type	CORBA Typedef Name	PL/I Representation
unsigned long long	CORBA-unsigned-long-long	FIXED BIN(31)[a,b]
wchar	CORBA-wchar	GRAPHIC

a. `UNSIGNED FIXED BIN` is not supported by the PL/I compiler for MVS & VM. Therefore, the maximum length of a PL/I `unsigned short` is half that of the CORBA-defined equivalent. The same applies for a PL/I `unsigned long` CORBA type.

b. The maximum number of digits allowed in a `FIXED BIN` is 31 bits.

Example

The example can be broken down as follows:

1. Consider the following IDL, stored in an IDL member called `EXAMPLE`:

```
const float  outer_float   =  19.76;
const double outer_double  =  123456.789;

interface example {
    typedef   fixed<5,2>          fixed_5_2;

    attribute short               myshort;
    attribute long                mylong;
    attribute unsigned short      ushort;
    attribute unsigned long       ulong;
    attribute float               myfloat;
    attribute double              mydouble;
    attribute char                mychar;
    attribute octet               myoctet;
    attribute fixed_5_2           myfixed52;
    attribute long long           mylonglong;
    attribute unsigned long long  ulonglong;

    const short   intf_sh   =   24;
    const wchar   mywchar   =   L'X';
    const wstring mywstring =   L"Hello";
};

module extras {
    const long    elong  =  760224;
};
```

2. The preceding IDL maps to the following in the *idlmembername*`M` include member:

```
/*-----------------------------------------------------*/
/* Constants in root scope:                             */
/*-----------------------------------------------------*/
dcl 1 global_EXAMPLE_consts,
    3 outer_float      float dec(6)   init(1.976e+01),
    3 outer_double     float dec(16)  init(1.23456789e+05);

/*-----------------------------------------------------*/
/* Constants in example:                                */
/*-----------------------------------------------------*/
dcl 1 example_consts,
    3 intf_sh          fixed bin(15)  init(24),
    3 mywchar          graphic(01)    init(graphic('X')),
    3 mywstring        graphic(05)    init(graphic('Hello'));

/*-----------------------------------------------------*/
/* Constants in extras:                                 */
/*-----------------------------------------------------*/
dcl 1 extras_consts,
    3 elong              fixed bin(31)  init(760224),
```

The *idlmembername*`M` include member also declares storage for the attributes.

3. Based on the preceding IDL in point 1, the definitions for the attributes are generated in the *idlmembername*`T` include member as follows (where generated comments have been omitted for the sake of brevity):

```
dcl 1 example_myshort_type based,
    3 result                   fixed bin(15)  init(0);

dcl 1 example_mylong_type_based,
    3 result                   fixed bin(31)  init(0);

dcl 1 example_ushort_type based,
    3 result                   fixed bin(15)  init(0);

dcl 1 example_ulong_type based,
    3 result                   fixed bin(31)  init(0);

dcl 1 example_myfloat_type_based,
    3 result                   float dec(6)   init(0.0);

dcl 1 example_mydouble_type based,
    3 result                   float bin(16)  init(0.0);

dcl 1 example_mychar_type based,
    3 result                   char(01)       init('');

dcl 1 example_myoctet_type_based,
    3 result                   char(01)       init(low(1));

dcl 1 example_myfixed52_type based,
    3 result                   fixed dec(5,2) init(0);

dcl 1 example_mylonglong_type based,
    3 result                   fixed bin(31)  init(0);

dcl 1 example_ulonglong_type_based,
    3 result                   fixed bin(31)  init(0);
```

Mapping for Boolean Type

Overview

This section describes how booleans are mapped to PL/I.

IDL-to-PL/I mapping for booleans

An IDL boolean type maps to a PL/I character data item. Two named constants representing the true and false values are provided.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface examp1e {
    attribute boolean full;
};
```

2. The preceding IDL maps to the following PL/I:

```
/* Declared in the Orbix PL/I CORBA include file */
DCL CORBA_FALSE            CHAR(01)  INIT('0') STATIC;
DCL CORBA_TRUE             CHAR(01)  INIT('1') STATIC;

/* Generated output by the IDL compiler */
dcl 1 example_full_type_based,
    3 result               char(01)   init(CORBA_FALSE);
```

Mapping for Enum Type

Overview

This section describes how enums are mapped to PL/I.

IDL-to-PL/I mapping for enums

An IDL enum type maps to PL/I `FIXED BIN(31) BINARY` named constants that are assigned an incrementing value starting from 0.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface weather {
    enum temp {cold, warm, hot};
};
```

2. The preceding IDL maps to the following PL/I:

```
/*------------------------------------------------------*/
/* Enum values in weather/temp:                         */
/*------------------------------------------------------*/
dcl weather_temp_cold     fixed bin(31)  init(0)  static;
dcl weather_temp_warm     fixed bin(31)  init(1)  static;
dcl weather_temp_hot      fixed bin(31)  init(2)  static;
```

3. It can be used as follows:

```
if todays_temp = weather_temp_cold then
    put skip list('Brr, it is cold outside!');
```

Mapping for Octet and Char Types

Overview

This section describes how octet and char types are mapped to PL/I.

IDL-to-PL/I mapping for char types

Char data values that are passed between machines with different character encoding methods (for example, ASCII, EBCDIC, and so on) are appropriately converted. See "Example" on page 184 for an example of how char types are mapped to PL/I.

IDL-to-PL/I mapping for octet types

Octet data values that are passed between machines with different character encoding methods (for example, ASCII, EBCDIC, and so on) are not converted. See "Example" on page 184 for an example of how octet types are mapped to PL/I.

Mapping for String Types

Overview

This section describes how string types are mapped to PL/I. First, it describes the various string types that are available.

Bounded and unbounded strings

Strings can be bounded or unbounded. Bounded strings are of a specified size, while unbounded strings have no specified size. For example:

```
//IDL
string<8>  a_bounded_string
string     an_unbounded_string
```

Bounded and unbounded strings are represented differently in PL/I. The maximum length of a bounded string in PL/I is 32,767 characters.

Incoming bounded strings

Incoming strings are passed as `IN` or `INOUT` values by the `PODGET` function into the PL/I operation parameter buffer at the start of a PL/I operation.

An incoming bounded string is represented by a `CHAR(`*n*`)` data item, where *n* is the bounded length of the string. Such strings have their nulls converted to spaces, if they contain nulls.

Outgoing bounded strings

Outgoing strings are copied as `INOUT`, `OUT`, or `RESULT` values by the `PODPUT` function from the complete PL/I operation parameter buffer that is passed to it at the end of a PL/I operation.

An outgoing bounded string has trailing spaces removed, and all characters up to the bounded length (or the first null) are passed via `PODPUT`. If a null is encountered before the bounded length, only those characters preceding the null are passed. The remaining characters are not passed.

Incoming unbounded strings

Incoming strings are passed as `IN` or `INOUT` values by the `PODGET` function into the PL/I operation parameter buffer at the start of a PL/I operation.

An incoming unbounded string is represented as a pointer data item. A pointer is supplied that refers to an area of memory containing the string data. This string is not directly accessible. You must call the `STRGET` function to copy the data into a `CHAR(`*n*`)` data item, because the length of the unbounded string is not known in advance. For example:

```
/* This is the supplied PL/I unbounded string pointer.   */
dcl name ptr;

/* This is the PL/I representation of the string.        */
dcl supplier_name char (64);

/* This STRGET call copies the characters in NAME to     */
/* SUPPLIER_NAME                                         */
call strget(name,supplier_name,length(supplier_name));
```

If the unbounded string that is passed is too big for the supplied PL/I string, an exception is raised and the PL/I string remains unchanged. If the unbounded string is not big enough to fill the PL/I string, the rest of the PL/I string is filled with spaces.

Outgoing unbounded strings

Outgoing strings are copied as `INOUT`, `OUT`, or `RESULT` values by the `PODPUT` function from the complete PL/I operation parameter buffer that is passed to it at the end of a PL/I operation.

A valid outgoing unbounded string must be supplied by the implementation of an operation. This can be either a pointer that was obtained by an `IN` or `INOUT` parameter, or a string constructed by using the `STRSET` function. For example:

```
/* This is the PL/I representation of the string containing a */
/* value that we want to pass back to the client using PODPUT */
/* via an unbounded pointer string.                           */
dcl notes char (160);

/* This is the unbounded pointer string                       */
dcl cust_notes ptr;

/* This STRGET call creates a copy of the string in the NOTES */
/* field and assigns the pointer value to                     */
call strset(cust_notes,notes,length(notes));
```

Trailing spaces are removed from the constructed string. If trailing spaces are required, you can use the STRSETS function, with the same argument signature, to copy the specified number of characters, including trailing spaces.

Example

The following is an example of how strings are mapped to PL/I. The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    attribute string mystring;
    string<10>getname(in string code);
};
```

2. The Orbix IDL compiler generates the following PL/I, based on the preceding IDL:

```
/*-------------------------------------------------------*/
/* Attribute:    mystring                                */
/* Mapped name:  mystring                                */
/* Type:         string (read/write)                     */
/*-------------------------------------------------------*/
dcl 1 example_mystring_type based,
    3 result                  ptr       init(sysnull());

/*-------------------------------------------------------*/
/* Operation:    getname                                 */
/* Mapped name:  getname                                 */
/* Arguments:    <in> string code                        */
/* Returns:      string<10>                              */
/*-------------------------------------------------------*/
dcl 1 example_getname_type based,
    3 code                    ptr       init(sysnull()),
    3 result                  char(10)  init('');
```

Mapping for Fixed Type

Overview

This section describes how fixed types are mapped to PL/I.

IDL-to-PL/I mapping for fixed types

The IDL fixed type maps directly to PL/I packed decimal data with the appropriate number of digits and decimal places (if any).

Fixed-point decimal data type

The fixed-point decimal data type is used to express in exact terms numeric values that consist of both an integer and a fixed-length decimal fraction part. The fixed-point decimal data type has the format `<d,s>`.

Examples of the fixed-point decimal data type

You might use it to represent a monetary value in dollars. For example:

```
fixed<9,2> net_worth; // up to $9,999,999.99, accurate to one
   cent
fixed<9,4> exchange_rate; // accurate to 1/10000 unit
fixed<4,-6> annual_revenue; // in millions
```

Explanation of the fixed-point decimal data type

The format of the fixed-point decimal data type can be explained as follows:

1. The first number within the angle brackets is the total number of digits of precision.
2. The second number is the scale (that is, the position of the decimal point relative to the digits).

   A positive scale represents a fractional quantity with that number of digits after the decimal point. A zero scale represents an integral value. A negative scale is allowed, and it denotes a number with units in positive powers of ten (that is, hundreds, millions, and so on).

Example of IDL-to-PL/I mapping for fixed types

The example can be broken down as follows:

1. Consider the following IDL:

```
//IDL
interface exam1e {
    typedef fixed<5,2>       typesal;
    typdef fixed<4,4>        typetax;
    typedef fixed<3,-6>      typemill;
    typedef fixed<6,3>       typesmall;
    attribute typesal salary;
    attribute typetax taxrate;
    attribute typemill millions;
    attribute typesmall small;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code (where comments are omitted for the sake of brevity):

```
dcl 1 example_salary_type based,
    3 result                fixed dec(5,2)  init(0);

dcl 1 example_taxrate_type based,
    3 result                fixed dec(4,4)  init(0);

dcl 1 example_millions_type based,
    3 result                fixed dec(3,-6) init(0);

dcl 1 example_small_type based,
    3 result                fixed dec(6,3)  init(0);
```

3. If you try to display a number such as `example_millions_args` or `example_small_args` (each of the identifiers with an `_args` suffix is declared as being `like` the `based` variables shown in point 2), the number is displayed as a floating point number; however, it is stored in the normal fixed format. The following example illustrates this point:

```
example_salary_args.result=165.78;
example_taxrate_args.result=0.9876;
example_millions_args.result=23000000;
example_small_args.result=0.041;

put skip list('Salary   =', example_salary_args.result);
put skip list('TaxRate  =', example_taxrate_args.result);
put skip list('Millions =', example_millions_args.result);
put skip list('Small    =', example_small_args.result);
```

4. Displaying the contents of each variable based on the preceding statements then produces the following:

```
Salary   =       165.78
TaxRate  =      0.9876
Millions =       23F+6
Small    =         0.004
```

Note: The maximum number of figures (not significant digits) allowed is 15. For example, `fixed<15,3>` is allowed; however, `fixed<15,-3>` is not allowed, because the number of digits required to display that number is 18.

Mapping for Struct Type

Overview

This section describes how struct types are mapped to PL/I.

IDL-to-PL/I mapping for struct types

An IDL struct definition maps directly to a PL/I structure.

Example of IDL-to-PL/I mapping for struct types

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    struct mystruct {
        long            member1;
        short           member2;
        boolean         member3;
    };
    attribute mystruct test;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following PL/I code for the `test` attribute:

```
dcl 1 example_test_type based,
    3 result,
       5 member1         fixed bin(31)    init(0),
       5 member2         fixed bin(15)    init(0),
       5 member3         char(01)         init(CORBA_FALSE);
```

Mapping for Union Type

Overview

This section describes how union types are mapped to PL/I.

IDL-to-PL/I mapping for union types

An IDL union maps to a PL/I structure that contains:

- A discriminator, d.
- The union data area, u.
- A PL/I structure for each union branch.

Example of IDL-to-PL/I mapping for union types

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    union un switch(short) {
        case 1: char        case_1;
        case 2: double      case_2;
        default: long       def_case;
    };
    attribute un test;
};
```

2. Based on the preceding IDL, the definition for the attribute's structure is generated as follows in the *idlmembername*T include member:

```
dcl 1 example_test_type based,
    3 result,
        5 d              fixed bin(15)  init(0),
        5 u              area(08);
```

The actual storage for the `test` attribute is generated as follows in the *idlmembername*M include member:

```
/*-------------------------------------------------------*/
/* Attribute:    test                                    */
/* Mapped name:  test                                    */
/* Type:         example/un (read/write)                 */
/*-------------------------------------------------------*/
dcl 1 example_test_attr aligned like example_test_type;
```

The union branches are generated as follows in the *idlmembername*M include member:

```
/*-----------------------------------------------------------*/
/* Initialization Statements for Union:                      */
/*    example/un                                             */
/*                                                           */
/* Used In:                                                  */
/*    example_test_attr.result                               */
/*-----------------------------------------------------------*/
dcl example_test_result_case_1 based(example_test_attr.result.u)
                               char(01)       init('');
dcl example_test_result_case_2 based(example_test_attr.result.u)
                               float dec(16)  init(0.0);
dcl example_test_result_def_case
   based(example_test_attr.result.u)
                               fixed bin(31)  init(0);
```

Compiler restrictions

Because the PL/I for MVS & VM compiler does not support unions directly, the union branches (in the preceding example, `case_1`, `case_2`, and `def_case`) are declared separately from the union structure. The union branches use the storage defined by the `example_test_attr.u` pseudo-union branch. This branch is allocated enough storage for the largest union item. In the preceding example, the largest union item is `case_2`, which is a `float dec (16)` type, thus requiring 8 bytes of storage.

Using the union type

To use the union type, for example, to display the contents retrieved by calling `get` on the attribute, you can use a `select` statement as follows:

```
select(example_test_attr.d)
    when(1)
        put skip list('Value of case_1 is:',
            example_test_result_case_1);
    when(2)
        put skip list('Value of case_2:',
            example_test_result_case_2);
    otherwise
        put skip list('Value of def_case is:',
            example_test_result_def_case);
end;
```

Setting up the attribute

You can set up the `test` attribute as follows, for example, to set up the value for the `get` call on the attribute (which is taken from the *idlmembername*`I` server implementation module):

```
/*-------------------------------------------------------------*/
/* Attribute:    test (get)                                    */
/* Mapped name:  test                                          */
/* Type:         example/un (read/write)                       */
/*-------------------------------------------------------------*/
proc_example_get_test: PROC(p_args);

dcl p_args                          ptr;
dcl 1 args                          aligned based(p_args)
                                    like example_test_type;

/* ============ Start of operation specific code ============ */
args.d=1;  /* case_1 */
example_test_result_case_1='Z';
/* ============= End of operation specific code ============= */

END proc_example_get_test;
```

Mapping for Sequence Types

Overview

The PL/I mapping for a sequence differs depending on whether the sequence is bounded or unbounded. In both cases, however, a supporting pointer that contains information about the sequence is generated. This information includes the maximum length (accessed via `SEQMAX`), the length of the sequence in elements (accessed via `SEQLEN`), and the contents of the sequence (in the case of the unbounded sequence). After a sequence is initialized, the sequence length is equal to zero. The first element of a sequence is referenced as element `1`. The `_dat` suffix contains the actual sequence data.

Bounded

Bounded sequence types map to a PL/I array and a supporting data item. For example:

```
interface example {
    typedef             sequence<long, 10>  seqlong10;
    attribute           seqlong10 myseq;
};
```

The preceding IDL maps to the following PL/I:

```
dcl 1 example_myseq_type based,
    3 result,
        5 result_seq          ptr            init(sysnull()),
        5 result_dat(10)      fixed bin(31)  init((10)0);
```

Unbounded

Unbounded sequence types cannot map to a PL/I array, because the size of the sequence is not known. In this case, a group item is created to hold one element of the sequence, and the element is provided with a suffix of `_buf`. A supporting pointer to the elements of the sequence is also created. For example:

```
interface example {
    typedef             sequence<long>  seqlong;
    attribute           seqlong         myseq;
};
```

The preceding IDL maps to the following PL/I:

```
dcl 1 example_myseq_type based,
    3 result,
        5 result_seq           ptr            init(sysnull()),
        5 result_buf           fixed bin(31)  init(0);
```

Initial storage is assigned to the sequence via `SEQALOC`. Elements of an unbounded sequence are not directly accessible. You can use `SEQGET` and `SEQSET` to access specific elements in the sequence. You can use `SEQLEN` to find the length of the sequence. You can use `SEQMAX` to find the maximum length of the sequence.

PODGET—IN and INOUT modes

An unbounded sequence is represented as a pointer data item. A pointer is supplied that refers to an area of memory containing the sequence. This is not directly accessible. You must call the `SEQGET` function to copy a specified element of the sequence into an accessible data area.

The following PL/I, based on the preceding IDL example, walks through all the elements of a sequence:

```
/* Excerpt from the M-suffixed include file:                   */
dcl 1 example_myseq_attr aligned like example_myseq_type;

/* Code for traversing through the unbounded sequence of longs */
dcl element_num     fixed bin(31)    init(0);
dcl result_seq      fixed bin(31)    init(0);

call seqlen(example_myseq_args.result.result_seq,
    result_seq_len);

do element_num = 1 to result_seq_len;
    call seqget(example_myseq_args.result.result_seq,
        element_num,
        addr(example_myseq_args.result.result_buf));
    put skip list('Element #',
        element_num,
        ' contains value',
        example_myseq_args.result.result_buf);
end;
```

PODPUT—OUT, INOUT, and result only

A valid unbounded sequence must be supplied by the implementation of an operation. This can be either a pointer that was obtained by an `IN` or `INOUT` parameter, or an unbounded sequence constructed by using the `SEQALOC` function.

The `SEQSET` function is used to change the contents of a sequence element. Based on the preceding example, the following code could be used to store some initial values into all elements of the sequence.

The following example uses the attribute defined in the preceding IDL for setting up the unbounded sequence of `long` types (note the `example_seqlong_tc` is the sequence typecode, which is declared in the *`idlmembername`*`T` include member):

```
dcl seq_size           fixed bin(31)   init(20);
del element_num        fixed bin(31)   init(0);

call seqlen(result_seq,result_seq_len);
call seqaloc(example_myseq_args.result.result_seq, seq_size,
    example_seqlong_tc, length(example_seqlong_tc);

do element_num = 1 to seq_size;
    result_buf=7*i;  /* 7 times multiplication table */
    call seqset(example_myseq_args.result.result_seq,
        element_num,
        addr(example_myseq_args.result.result_buf);
end;
```

Mapping for Array Type

Overview

This section describes how arrays are mapped to PL/I.

IDL-to-PL/I mapping for arrays

An IDL array definition maps directly to a PL/I array. Each element of the array is directly accessible.

> **Note:** PL/I arrays are `1`-indexed, and not `0`-indexed as in C or C++. For example, grid reference `A(1,2)` in PL/I matches `A[2][3]` in C++.

Example of IDL-to-PL/I mapping for arrays

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    typedef long mylong[2][5];
    attribute mylong long_array;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *idlmembername*`T` include member:

```
dcl 1 example_long_array_type based,
    3 result(2,5)             fixed bin(31)  init ((2*5)0);
```

The Orbix IDL compiler generates the following code in the *idlmembername*`M` include member:

```
dcl 1 example_long_array_attr aligned like
   example_long_array_type;
```

3. The following is an example of how the generated code can subsequently be used:

```
example_long_array_args.result(1,3) = 22;
```

Mapping for the Any Type

Overview

This section describes how anys are mapped to PL/I.

IDL-to-PL/I mapping for anys

The IDL any type maps to a PL/I structure that provides information about the contents of the any, such as the type of the contents. A separate character data item is also generated, which is large enough to hold the longest type code string defined in the interface.

Example of IDL-to-PL/I mapping for anys

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    typedef any myany;
    attribute myany temp;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *idlmembername*T include member:

```
dcl 1 example_temp_type based,
    3 result                 ptr           init(sysnull());

dcl EXAMPLE_typecode          char(21)       init('');
dcl example_myany_tc          char(21)
                              init('IDL:example/myany:1.0');
dcl EXAMPLE_typecode_length   fixed bin(31)  init(21);
```

In the preceding example, EXAMPLE_typecode is used as a variable when setting the type of the any. The typecode identifier for the any, which is used for sequences, is defined in the preceding example as example_myany_tc. The maximum length of all the typecodes defined in the IDL is 21, which is defined via EXAMPLE_typecode_length. In the preceding example, EXAMPLE denotes the IDL member name, and example denotes the interface name.

Accessing and changing contents of an any

You cannot access the contents of the any type directly. Instead you can use the ANYGET function to extract data from an any type, and use the ANYSET function to insert data into an any type.

Before you call ANYGET, call TYPEGET to retrieve the type of the any into a data item generated by the Orbix IDL compiler. This data item is large enough to hold the largest type name defined in the interface. Similarly, before you call ANYSET, call TYPESET to set the type of the any.

See "ANYGET" on page 317 and "TYPEGET" on page 409 for details and an example of how to access the contents of an any. See "ANYSET" on page 319 and "TYPESET" on page 412 for details and an example of how to change the contents of an any.

Mapping for User Exception Type

Overview

This section describes how exceptions are mapped to PL/I.

IDL-to-PL/I mapping for exceptions

An IDL exception type maps to a PL/I structure and a character data item with a value that uniquely identifies the exception.

Example of IDL-to-PL/I mapping for exceptions

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    exception bad {
        long       value1;
        string<32> reason;
    };

    exception worse {
        short      value2;
        string<16> errorcode;
        string<32> reason;
    };

    void addName(in string name) raises(bad,worse);
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *idlmembername*T include member:

```
dcl 1 example_addName_type based,
    3 idl_name             ptr            init(sysnull());
```

3. The Orbix IDL compiler generates the following code in the *idlmembername*M include member:

```
/*-------------------------------------------------------*/
/* Operation:    addName                                 */
/* Mapped name:  addName                                 */
/* Arguments:    <in> string name                        */
/* Returns:      void                                    */
/*-------------------------------------------------------*/
dcl 1 example_addName_args aligned like
   example_addName_type;

/*-------------------------------------------------------*/
/* Defined User Exceptions                               */
/*-------------------------------------------------------*/
dcl 1 EXAMPLE_user_exceptions,
    3 exception_id        ptr,
    3 d                   fixed bin(31)  init(0),
    3 u                   area(50);

dcl 1 example_bad_exc_d    fixed bin(31)  init(1);
dcl 1 example_worse_exc_d  fixed bin(31)  init(2);

dcl 1 example_bad_exc      based(EXAMPLE_user_exceptions.u),
    3 value1              fixed bin(31)  init(0),
    3 reason              char(32)       init('');

dcl 1 example_worse_exc    based(EXAMPLE_user_exceptions.u),
    3 value2              fixed bin(15)  init(0),
    3 errorcode           char(16)       init(''),
    3 reason              char(32)       init('');
```

Raising a user exception

The server can raise a user exception by performing the following sequence of steps:

1. It calls `STRSET` to set the `exception_id` identifier of the user exception structure with the appropriate exception identifier defined in the *idlmembername*T include member. The exception identifier in this case is suffixed with `_exid`.
2. It sets the `d` discriminator with the appropriate exception identifier defined in the *idlmembername*M include member. The exception identifier in this case is suffixed with `_d`.
3. It fills in the exception branch block associated with the exception.

4. It calls `PODERR` with the address of the user exception structure.

Example of Error Raising and Checking

The example can be broken down as follows:

1. The following code shows how to raise the `bad` user exception defined in the preceding example:

```
/* Server implementation code  */
if name='' then
    do;
        strset(EXAMPLE_user_exceptions.exception_id,
               SimpleObject_bad_exid,
               length(SimpleObject_bad_exid));
        EXAMPLE_user_exceptions.d=example_bad_exc_d;
        call poderr(addr(EXAMPLE_user_exceptions));
    end;
```

2. To test for the user exception, the client side tests the discriminator value of the user exception structure after calling `PODEXEC` on the server function, which is able to raise a user exception. For example, the following code shows how the client can test whether the server set an exception after the call to `addName`:

Example 8: *Client Code to Test Exception (Sheet 1 of 2)*

```
/* Call podexec to perform operation addName.             */
/* Note the user exception block in the fourth parameter. */
call podexec(example_obj,
             example_addName,
             example_addName_args,
             addr(EXAMPLE_user_exceptions));

if EXAMPLE_user_exceptions.d ^= 0 then
   do;
      /* a user exception has been thrown */
      put skip list('Operation addName threw a user exception!');
      put skip list(' Discriminator: ',EXCEPT_user_exceptions.d);

      select(EXAMPLE_user_exceptions.d);
         when(example_bad_exc_d)
            do;
               put list('Exception thrown: bad_exc');
               put skip list('value1:',example_bad_exc.value1);
               put skip list('reason:',example_bad_exc.reason);
            end;
```

Example 8: *Client Code to Test Exception (Sheet 2 of 2)*

```
         when(example_worse_exc_d)
            do;
               put list('Exception thrown: worse_exc');
               put skip list('value2:',example_worse_exc.value2);
               put skip list('errorcode:',
                  example_worse_exc.errorcode);
               put skip list('reason:',example_worse_exc.reason);
            end;
         otherwise
            put list('Unrecognized exception!');
      end;
   end;
else  /* no exception has been thrown  */
   do;
      put skip list('Operation addName completed successfully');
   end;
```

Mapping for Typedefs

Overview

This section describes how typedefs are mapped to PL/I.

IDL-to-PL/I mapping for typedefs

Typdefs are supported in PL/I through the use of the `based` keyword. The Orbix IDL compiler generates `based` declarations for attribute and operation structures (to keep them generic), for struct types, and for other complex types. It does not generate a based identifier in a one-to-one mapping with the IDL unless all of the typedefs defined in the IDL are these types just listed.

The reasons for this are partially to do with how the PL/I runtime uses them to set up and retrieve data, and partially for ease of coding. In the case of ease-of-coding, if an operation has two parameters, but is then changed to have three parameters, only the `based` declaration needs to be updated, because each of the uses of the particular operation are declared as being `like` the `based` structure.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    typedef struct stru;
        long a;
        short b;
    } misc;
    typedef fixed<8,2> currency;

    attribute currency pounds;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates a `based` identifier for the struct, `stru`, and for the attribute structure; however, it does not generate a `based` identifier for the fixed type. The `based` variables for the struct, `stru`, are generated in the *`idlmembername`*`T` include member as follows:

```
dcl 1 example_pounds_type based,
    3 result                fixed dec(8,2)  init(0);

/*------------------------------------------------------*/
/* Struct: example/stru                                 */
/*------------------------------------------------------*/
dcl 1 example_stru_type based,
    3 a                     fixed bin(31)   init(0),
    3 b                     fixed bin(15)   init(0);
```

3. The attribute's structure is generated as follows in the *`idlmembername`*`M` include member, which makes use of the attribute's based structure:

```
dcl 1 example_pounds_attr aligned like example_pounds_type;
```

Mapping for Operations

Overview

This section describes how operations are mapped to PL/I.

IDL-to-PL/I mapping for operations

An IDL operation maps to a number of statements in PL/I as follows:

1. A structure is created for each operation. This structure is declared in the *idlmembername*T include member as a based structure and contains a list of the parameters and the return type of the operation. An associated declaration, which uses this based structure, is declared in the *idlmembername*M include member. Memory is allocated only for non-dynamic types, such as bounded strings and longs. The top-level identifier (that is, at dcl 1 level) for each operation declaration is suffixed with _type in the *idlmembername*T include member, and with _args in the *idlmembername*M include member, for example:

```
dcl 1 my_operation_type_based,
    3 my_argument  fixed bin(31) init(0);
    …
```

2. A declaration is generated in the *idlmembername*T include member for every IDL operation. The declaration contains the fully qualified operation name followed by a space, which is used when calling PODEXEC to invoke that operation on a server. The following is an example of a declaration based on the my_operation operation in the test interface:

```
dcl test_my_operation  char(36)
    init('my_operation:IDL:test:1.0 ');
```

3. The operation declaration is also used in the *idlmembername*D include member. It is used within the select clause, which is used by the server program to call the appropriate operation/attribute procedure described next in point 4.

4. When you specify the `-S` argument with the Orbix IDL compiler, an empty server procedure is generated in the *idlmembername*`I` source member for each IDL operation. (You must specify the `-S` argument, to generate these operation/attribute procedures.)

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example
{
    long my_operation1(in long mylong);
    short my_operation2(in short myshort);
};
```

2. Based on the preceding IDL, the following operation structures are generated in the *idlmembername*`T` include member:

```
/*-----------------------------------------------------------*/
/* Operation:       my_operation1                           */
/* Mapped name:     my_operation1                           */
/* Arguments:       <in> long mylong                        */
/* Returns:         long                                    */
/* User Exceptions: none                                    */
/*-----------------------------------------------------------*/
dcl 1 example_my_operation1_type based,
      3 mylong                    fixed bin(31)  init(0),
      3 result                    fixed bin(31)  init(0);

/*-----------------------------------------------------------*/
/* Operation:       my_operation2                           */
/* Mapped name:     my_operation2                           */
/* Arguments:       <in> short myshort                      */
/* Returns:         short                                   */
/* User Exceptions: none                                    */
/*-----------------------------------------------------------*/
dcl 1 example_my_operation2_type based,
    3 myshort                     fixed bin(15)  init(0),
    3 result                      fixed bin(15)  init(0);
```

3. Based on the preceding IDL, the following operation structures are generated in the *idlmembername*M include member:

```
/*------------------------------------------------------*/
/* Operation:       my_operation1                       */
/* Mapped name:     my_operation1                       */
/* Arguments:       <in> long mylong                    */
/* Returns:         long                                */
/* User Exceptions: none                                */
/*------------------------------------------------------*/
dcl 1 example_my_operation1_args aligned like
      example_my_operation1_type;

/*------------------------------------------------------*/
/* Operation:       my_operation2                       */
/* Mapped name:     my_operation2                       */
/* Arguments:       <in> short myshort                  */
/* Returns:         short                               */
/* User Exceptions: none                                */
/*------------------------------------------------------*/
dcl 1 example_my_operation2_args aligned like
      example_my_operation2_type;
```

4. The following is generated in the *idlmembername*T include member:

```
/*------------------------------------------------------*/
/* Operation List section                               */
/* Contains a list of the interface's operations and    */
/* attributes.                                          */
/*------------------------------------------------------*/
dcl example_my_operation1    char(30)
    init('my_operation1:IDL:example:1.0 ');
dcl example_my_operation2    char(30)
    init('my_operation2:IDL:example:1.0 ');
```

5. The following `select` statement is also generated in the *idlmembername*`D` include member:

```
select(operation);
     when (example_my_operation1) do;
         call podget(addr(example_my_operation1_args));
         if check_errors('podget') ^= completion_status_yes
             then return;

         call proc_example_my_operation1
             (addr(example_my_operation1_args));

         call podput(addr(example_my_operation1_args));
         if check_errors('podput') ^= completion_status_yes
             then return;
     end;
     when (example_my_operation2) do;
         call podget(addr(example_my_operation2_args));
         if check_errors('podget') ^= completion_status_yes
             then return;

         call proc_example_my_operation2
             (addr(example_my_operation2_args));

         call podput(addr(example_my_operation2_args));
         if check_errors('podput') ^= completion_status_yes
             then return;
     end;
     otherwise do;
         put skip list('ERROR! Undefined Operation ' ||
             operation);
         return;
     end;
end;
```

6. The following skeleton procedures are generated in the *idlmembername*`I` member:

```
/*-------------------------------------------------------*/
/* Operation:       my_operation1                         */
/* Mapped name:     my_operation1                         */
/* Arguments:       <in> long mylong                      */
/* Returns:         long                                  */
/* User Exceptions: none                                  */
/*-------------------------------------------------------*/
proc_example_my_operation1: PROC(p_args);

dcl p_args                    ptr;
dcl 1 args                    aligned based(p_args)
                            like example_my_operation1_type;

/* ========= Start of operation specific code ========== */
/* ========== End of operation specific code =========== */

END proc_example_my_operation1;

/*-------------------------------------------------------*/
/* Operation:       my_operation2                         */
/* Mapped name:     my_operation2                         */
/* Arguments:       <in> short myshort                    */
/* Returns:         short                                 */
/* User Exceptions: none                                  */
/*-------------------------------------------------------*/
proc_example_my_operation2: PROC(p_args);

dcl p_args                     ptr;
dcl 1 args                     aligned based(p_args)
                            like example_my_operation2_type;

/* =========== Start of operation specific code ======= */
/* ============ End of operation specific code ========= */

END proc_example_my_operation2;
```

Mapping for Attributes

Overview

This section describes how IDL attributes are mapped to PL/I.

Similarity to mapping for operations

The IDL mapping for attributes is very similar to the IDL mapping for operations, but with the following differences:

- IDL attributes map to PL/I with a `_get_` and `_set_` prefix. Two PL/I declarations are created for each attribute (that is, one with a `_get_` prefix, and one with a `_set_` prefix). However, readonly attributes only map to one declaration, with a `_get_` prefix.
- The top-level identifier (that is, at `dcl 1 level`) for each attribute declaration in the *idlmembername*`M` include member has a suffix of `_attr` rather than a suffix of `_args`.
- An attribute's parameters are always treated as return types (that is, a structure created for a particular attribute always contains just one immediate sub-declaration, `result`).

IDL-to-PL/I mapping for attributes

An IDL attribute maps to a number of statements in PL/I as follows:

1. A structure is created for each attribute. This structure is declared in the *idlmembername*`T` include member as a `based` structure and contains one immediate sub-declaration, `result`. If the attribute is a complex type, the `result` declaration contains a list of the attribute's parameters as lower-level declarations. If the parameters are of a dynamic type (for example, sequences, unbounded strings, or anys), no storage is assigned to them. An associated declaration, which uses this `based` structure, is declared in the *idlmembername*`M` include member.

   The top-level identifier (that is, at `dcl 1` level) for each attribute declaration is suffixed with `_type` in the *idlmembername*`T` include member, and with `_attr` in the *idlmembername*`M` include member (that is, *`FQN_attributename`*`_type` and *`FQN_attributename`*`_attr`).

2. Two declarations are generated in the *idlmembername*T include member for every IDL attribute, unless it is a readonly attribute, in which case only one declaration is declared for it. A declaration contains the fully qualified name followed by `_get_` or (provided it is not readonly) `_set_`, followed by the attribute name, followed by a space, which is used when calling `PODEXEC` to invoke that attribute on on a server. For example, the following is an example of two declarations based on the `myshort` attribute in the `example` interface:

```
dcl example_get_myshort  char(29)
    init('_get_myshort:IDL:example:1.0 ');
dcl example_set_myshort char (29)
    init('_set_myshort:IDL:example:1.0 ');
```

3. The attribute declaration is also used in the *idlmembername*D include member. It is used within the `select` clause, which is used by the server program to call the appropriate operation/attribute procedure described next in point 4.
4. When you specify the `-S` argument with the Orbix IDL compiler, an empty server procedure is generated in the *idlmembername*I source member for each IDL attribute. (You must specify the `-S` argument, to generate these operation/attribute procedures.)

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example
{
    readonly attribute long mylong;
    attribute short myshort;
};
```

2. Based on the preceding IDL, the following attribute structures are generated in the *idlmembername*T include member:

```
/*-----------------------------------------------------------*/
/* Attribute:       mylong                                   */
/* Mapped name:     mylong                                   */
/* Type:            long (readonly)                          */
/*-----------------------------------------------------------*/
dcl 1 example_mylong_type based,
      3 result                    fixed bin(31)  init(0),

/*-----------------------------------------------------------*/
/* Attribute:       myshort                                  */
/* Mapped name:     myshort                                  */
/* Type:            short (read/write)                       */
/*-----------------------------------------------------------*/
dcl 1 example_myshort_type based,
      3 result                      fixed bin(15)  init(0);
```

3. Based on the preceding IDL, the following attribute structures are generated in the *idlmembername*M include member:

```
/*-----------------------------------------------------------*/
/* Attribute:       mylong                                   */
/* Mapped name:     mylong                                   */
/* Type:            long (readonly)                          */
/*-----------------------------------------------------------*/
dcl 1 example_mylong_attr aligned like example_mylong_type;

/*-----------------------------------------------------------*/
/* Attribute:       myshort                                  */
/* Mapped name:     myshort                                  */
/* Type:            short (read/write)                       */
/*-----------------------------------------------------------*/
dcl 1 example_myshort_attr aligned like exampl_myshort_type;
```

4. The following is generated in the *idlmembername*T include member:

```
/*------------------------------------------------------------*/
/* Operation List section                                     */
/* Contains a list of the interface's operations and          */
/* attributes.                                                */
/*------------------------------------------------------------*/
dcl example_get_mylong    char(28)
    init('_get_mylong:IDL:example:1.0 ');
dcl example_get_myshort   char(29)
    init('_get_myshort:IDL:example:1.0 ');
dcl example_set_myshort   char(29)
    init('_set_myshort:IDL:example:1.0 ');
```

5. The following `select` statement is also generated in the *idlmembername*D include member:

```
select(operation);
     when (example_get_mylong) do;
         call podget(addr(example_mylong_attr));
         if check_errors('podget') ^= completion_status_yes
             then return;

         call proc_example_get_mylong
             (addr(example_mylong_attr));

         call podput(addr(example_mylong_attr));
         if check_errors('podput') ^= completion_status_yes
             then return;
     end;
     when (example_get_myshort) do;
         call podget(addr(example_myshort_attr));
         if check_errors('podget') ^= completion_status_yes
             then return;

         call proc_example_get_myshort
             (addr(example_myshort_attr));

         call podput(addr(example_myshort_attr));
         if check_errors('podput') ^= completion_status_yes
             then return;
     end;
```

```
     when (example_set_myshort) do;
         call podget(addr(example_myshort_attr));
         if check_errors('podget') ^= completion_status_yes
             then return;

         call proc_example_set_myshort
             (addr(example_myshort_attr));

         call podput(addr(example_myshort_attr));
         if check_errors('podput') ^= completion_status_yes
             then return;
     end;
     otherwise do;
         put skip list('ERROR! No such operation:')
         put skip list(operation);
         return;
     end;
end;
```

6. The following skeleton procedures are generated in the *idlmembername*I include member:

```
/*-------------------------------------------------------*/
/* Attribute:      mylong (get)                          */
/* Mapped name:    mylong                                */
/* Type:           long (readonly)                       */
/*-------------------------------------------------------*/
proc_example_get_mylong: PROC(p_args);

dcl p_args                    ptr;
dcl 1 args                    aligned based(p_args)
                              like example_mylong_type;

/* ========= Start of operation specific code ========== */
/* ========== End of operation specific code =========== */

END proc_example_get_mylong;
```

```
/*------------------------------------------------------*/
/* Attribute:       myshort (get)                      */
/* Mapped name:     myshort                            */
/* Type:            short (read/write)                 */
/*------------------------------------------------------*/
proc_example_get_myshort: PROC(p_args);

dcl p_args                     ptr;
dcl 1 args                     aligned based(p_args)
                               like example_myshort_type;

/* =========== Start of operation specific code ======= */
/* ============ End of operation specific code ========= */

END proc_example_get_myshort;

/*------------------------------------------------------*/
/* Attribute:       myshort (set)                      */
/* Mapped name:     myshort                            */
/* Type:            short (read/write)                 */
/*------------------------------------------------------*/
proc_example_set_myshort: PROC(p_args);

dcl p_args                     ptr;
dcl 1 args                     aligned based(p_args)
                               like example_myshort_type;

/* =========== Start of operation specific code ======= */
/* ============ End of operation specific code ========= */

END proc_example_set_myshort;

END EXAMPLI;
```

Mapping for Operations with a Void Return Type and No Parameters

Overview

This section describes IDL operations that have a void return type and no parameters are mapped to PL/I.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example
{
   void myoperation();
};
```

2. The preceding IDL maps to the following PL/I:

```
/*------------------------------------------------------*/
/* Operation:       myoperation                         */
/* Mapped name:     myoperation                         */
/* Arguments:       None                                */
/* Returns:         void                                */
/* User Exceptions: none                                */
/*------------------------------------------------------*/
dcl 1 example_myoperation_type based,
    3 filler_0001               char(01);
```

Note: The filler is included for completeness, to allow the application to compile, but the filler is never actually referenced. The numeric suffix can have any value. The other generated code segments are generated as expected.

Mapping for Inherited Interfaces

Overview

This section describes how inherited interfaces are mapped to PL/I.

IDL-to-PL/I mapping for inherited interfaces

An IDL interface that inherits from other interfaces includes all the attributes and operations of those other interfaces. In the header of the interface being processed, the Orbix IDL compiler generates an extra comment that contains a list of all the inherited interfaces.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface Account
{
   attribute short mybaseshort;
   void mybasefunc(in long mybaselong);
};

interface SavingAccount : Account
{
   attribute short myshort;
   void myfunc(in long mylong);
};
```

2. The preceding IDL is mapped to the following PL/I in the *idlmembername*D include member:

Example 9: *The idlmembernameD Example (Sheet 1 of 4)*

```
select(operation);
when (Account_get_mybaseshort) do;
    call podget(addr(Account_mybaseshort_attr));
    if check_errors('podget') ^= completion_status_yes
        then return;

    call proc_Account_get_mybaseshort
        (addr(Account_mybaseshort_attr));

    call podput(addr(Account_mybaseshort_attr));
    if check_errors('podput') ^= completion_status_yes
```

Example 9: *The idlmembernameD Example (Sheet 2 of 4)*

```
        then return;
end;
when (Account_set_mybaseshort) do;
    call podget(addr(Account_mybaseshort_attr));
    if check_errors('podget') ^= completion_status_yes
        then return;

    call proc_Account_set_mybaseshort
        (addr(Account_mybaseshort_attr));

    call podput(addr(Account_mybaseshort_attr));
    if check_errors('podput') ^= completion_status_yes
        then return;
end;
when (Account_mybasefunc) do;
    call podget(addr(Account_mybasefunc_args));
    if check_errors('podget') ^= completion_status_yes
        then return;

    call proc_Account_mybasefunc
        (addr(Account_mybasefunc_args));

    call podput(addr(Account_mybasefunc_args));
    if check_errors('podput') ^= completion_status_yes
        then return;
end;
when (SavingAccount_get_myshort) do;
    call podget(addr(SavingAccount_myshort_attr));
    if check_errors('podget') ^= completion_status_yes
        then return;

    call proc_SavingAccount_get_myshort
        (addr(SavingAccount_myshort_attr));

    call podput(addr(SavingAccount_myshort_attr));
    if check_errors('podput') ^= completion_status_yes
        then return;
end;
when (SavingAccount_set_myshort) do;
    call podget(addr(SavingAccount_myshort_attr));
    if check_errors('podget') ^= completion_status_yes
        then return;

    call proc_SavingAccount_set_myshort
        (addr(SavingAccount_myshort_attr));
```

Example 9: *The idlmembernameD Example (Sheet 2 of 4)*

```
        then return;
end;
when (Account_set_mybaseshort) do;
    call podget(addr(Account_mybaseshort_attr));
    if check_errors('podget') ^= completion_status_yes
        then return;

    call proc_Account_set_mybaseshort
        (addr(Account_mybaseshort_attr));

    call podput(addr(Account_mybaseshort_attr));
    if check_errors('podput') ^= completion_status_yes
        then return;
end;
when (Account_mybasefunc) do;
    call podget(addr(Account_mybasefunc_args));
    if check_errors('podget') ^= completion_status_yes
        then return;

    call proc_Account_mybasefunc
        (addr(Account_mybasefunc_args));

    call podput(addr(Account_mybasefunc_args));
    if check_errors('podput') ^= completion_status_yes
        then return;
end;
when (SavingAccount_get_myshort) do;
    call podget(addr(SavingAccount_myshort_attr));
    if check_errors('podget') ^= completion_status_yes
        then return;

    call proc_SavingAccount_get_myshort
        (addr(SavingAccount_myshort_attr));

    call podput(addr(SavingAccount_myshort_attr));
    if check_errors('podput') ^= completion_status_yes
        then return;
end;
when (SavingAccount_set_myshort) do;
    call podget(addr(SavingAccount_myshort_attr));
    if check_errors('podget') ^= completion_status_yes
        then return;

    call proc_SavingAccount_set_myshort
        (addr(SavingAccount_myshort_attr));
```

Example 9: *The idlmembernameD Example (Sheet 3 of 4)*

```
    call podput(addr(SavingAccount_myshort_attr));
    if check_errors('podput') ^= completion_status_yes
        then return;
end;
when (SavingAccount_myfunc) do;
    call podget(addr(SavingAccount_myfunc_args));
    if check_errors('podget') ^= completion_status_yes
        then return;

    call proc_SavingAccount_myfunc
        (addr(SavingAccount_myfunc_args));

    call podput(addr(SavingAccount_myfunc_args));
    if check_errors('podput') ^= completion_status_yes
        then return;
end;
when (SavingAccount_get_mybaseshort) do;
    call podget(addr(SavingAccount_mybaseshort_attr));
    if check_errors('podget') ^= completion_status_yes
        then return;

    call proc_SavingAccount_get_myb_dc3a
        (addr(SavingAccount_mybaseshort_attr));

    call podput(addr(SavingAccount_mybaseshort_attr));
    if check_errors('podput') ^= completion_status_yes
        then return;
end;
when (SavingAccount_set_mybaseshort) do;
    call podget(addr(SavingAccount_mybaseshort_attr));
    if check_errors('podget') ^= completion_status_yes
        then return;

    call proc_SavingAccount_set_myb_8e2b
        (addr(SavingAccount_mybaseshort_attr));

    call podput(addr(SavingAccount_mybaseshort_attr));
    if check_errors('podput') ^= completion_status_yes
        then return;
end;
when (SavingAccount_mybasefunc) do;
    call podget(addr(SavingAccount_mybasefunc_args));
    if check_errors('podget') ^= completion_status_yes
        then return;
```

Example 9: *The idlmembernameD Example (Sheet 4 of 4)*

```
    call proc_SavingAccount_mybasefunc
        (addr(SavingAccount_mybasefunc_args));

    call podput(addr(SavingAccount_mybasefunc_args));
    if check_errors('podput') ^= completion_status_yes
        then return;
end;
otherwise do;
    put skip list('ERROR! Undefined operation ' ||
        operation);
    return;
end;
end;
```

3. The following code is contained in the *idlmembername*T include member:

Example 10: *The idlmembernameT Example (Sheet 1 of 3)*

```
/*--------------------------------------------------------*/
/* Interface:                                             */
/*     Account                                            */
/*                                                        */
/* Mapped name:                                           */
/*     Account                                            */
/*                                                        */
/* Inherits interfaces:                                   */
/*     (none)                                             */
/*--------------------------------------------------------*/
/*--------------------------------------------------------*/
/* Attribute:    mybaseshort                              */
/* Mapped name:  mybaseshort                              */
/* Type:         short (read/write)                       */
/*--------------------------------------------------------*/
 dcl 1 Account_mybaseshort_type based,
     3 result                    fixed bin(15)  init(0);

/*--------------------------------------------------------*/
/* Operation:       mybasefunc                            */
/* Mapped name:     mybasefunc                            */
/* Arguments:       <in> long mybaselong                  */
/* Returns:         void                                  */
/* User Exceptions: none                                  */
```

Example 10: *The idlmembernameT Example (Sheet 2 of 3)*

```
/*------------------------------------------------------*/
 dcl 1 Account_mybasefunc_type based,
     3 mybaselong                fixed bin(31)  init(0);

/*------------------------------------------------------*/
/* Interface:                                           */
/*     SavingAccount                                    */
/*                                                      */
/* Mapped name:                                         */
/*     SavingAccount                                    */
/*                                                      */
/* Inherits interfaces:                                 */
/*     Account                                          */
/*------------------------------------------------------*/
/*------------------------------------------------------*/
/* Attribute:    myshort                                */
/* Mapped name:  myshort                                */
/* Type:         short (read/write)                     */
/*------------------------------------- ----------------*/
 dcl 1 SavingAccount_myshort_type based,
     3 result                    fixed bin(15)  init(0);
/*------------------------------------------------------*/
/* Attribute:    mybaseshort                            */
/* Mapped name:  mybaseshort                            */
/* Type:         short (read/write)                     */
/*------------------------------------------------------*/
 dcl 1 SavingAccount_mybaseshort_type based,
     3 result                    fixed bin(15)  init(0);

/*------------------------------------------------------*/
/* Operation:       myfunc                              */
/* Mapped name:     myfunc                              */
/* Arguments:       <in> long mylong                    */
/* Returns:         void                                */
/* User Exceptions: none                                */
/*------------------------------------------------------*/
 dcl 1 SavingAccount_myfunc_type based,
     3 mylong                    fixed bin(31)  init(0);

/*------------------------------------------------------*/
/* Operation:       mybasefunc                          */
/* Mapped name:     mybasefunc                          */
/* Arguments:       <in> long mybaselong                */
/* Returns:         void                                */
/* User Exceptions: none                                */
```

Example 10: *The idlmembernameT Example (Sheet 3 of 3)*

```
/*-------------------------------------------------------*/
 dcl 1 SavingAccount_mybasefunc_type based,
     3 mybaselong                fixed bin(31)  init(0);
/*-------------------------------------------------------*/
/* Operation List section                                */
/* Contains a list of the interface's operations and     */
/* attributes.                                           */
/*-------------------------------------------------------*/
 dcl Account_get_mybaseshort       char(33)
     init('_get_mybaseshort:IDL:Account:1.0 ');
 dcl Account_set_mybaseshort       char(33)
     init('_set_mybaseshort:IDL:Account:1.0 ');
 dcl Account_mybasefunc            char(27)
     init('mybasefunc:IDL:Account:1.0 ');
 dcl SavingAccount_get_myshort     char(35)
     init('_get_myshort:IDL:SavingAccount:1.0 ');
 dcl SavingAccount_set_myshort     char(35)
     init('_set_myshort:IDL:SavingAccount:1.0 ');
 dcl SavingAccount_myfunc          char(29)
     init('myfunc:IDL:SavingAccount:1.0 ');
 dcl SavingAccount_get_mybaseshort char(39)
     init('_get_mybaseshort:IDL:SavingAccount:1.0 ');
 dcl SavingAccount_set_mybaseshort char(39)
     init('_set_mybaseshort:IDL:SavingAccount:1.0 ');
 dcl SavingAccount_mybasefunc      char(33)
     init('mybasefunc:IDL:SavingAccount:1.0 ');
```

Mapping for Multiple Interfaces

Overview

This section describes how multiple interfaces are mapped to PL/I.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example1
{
    readonly attribute long mylong;
};

interface example2
{
    readonly attribute long mylong;
};
```

2. The *idlmembername*I member includes *idlmembername*D, to determine which server operation procedure is to be called. For example:

```
select(operation);
     when (example1_get_mylong) do;
         call podget(addr(example1_mylong_attr));
         if check_errors('podget') ^= completion_status_yes
             then return;

         call proc_example1_get_mylong
             (addr(example1_mylong_attr));

         call podput(addr(example1_mylong_attr));
         if check_errors('podput') ^= completion_status_yes
             then return;
     end;
     when (example2_get_mylong) do;
         call podget(addr(example2_mylong_attr));
         if check_errors('podget') ^= completion_status_yes
             then return;

         call proc_example2_get_mylong
             (addr(example2_mylong_attr));

         call podput(addr(example2_mylong_attr));
         if check_errors('podput') ^= completion_status_yes
             then return;
     end;
     otherwise do;
         put skip list('ERROR! Undefined operation ' ||
             operation);
         return;
     end;
end;
```

Orbix IDL Compiler

This chapter describes the Orbix IDL compiler in terms of how to run it in batch and OS/390 UNIX System Services, the PL/I source code and include members that it creates, the arguments that you can use with it, and the configuration variables that it uses.

In this chapter

This chapter discusses the following topics:

Note: The supplied demonstrations include examples of JCL that can be used to run the Orbix IDL compiler. You can modify the demonstration JCL as appropriate, to suit your applications. Any occurrences of *`orbixhlq`* in this chapter are meant to represent the high-level qualifier for your Orbix Mainframe installation on OS/390. If you are using OS/390 UNIX System Services, references to OS/390 member names can be interchanged with filenames, unless otherwise specified.

Running the Orbix IDL Compiler

Overview

You can use the Orbix IDL compiler to generate PL/I source code and include members from IDL definitions. This section describes how to run the Orbix IDL compiler, both in batch and in OS/390 UNIX System Services.

In this section

This section discusses the following topics:

Running the Orbix IDL Compiler in Batch

Overview

This subsection describes how to run the Orbix IDL compiler in batch. It discusses the following topics:

- "Orbix IDL compiler configuration" on page 235.
- "Running the Orbix IDL compiler" on page 235.
- "Example of the batch SIMPLIDL JCL" on page 236.
- "Description of the JCL" on page 237.

Orbix IDL compiler configuration

The Orbix IDL compiler uses the Orbix configuration member for its settings. The JCL that runs the compiler uses the `IDL` member in the *`orbixhlq`*`.CONFIG` configuration PDS.

Running the Orbix IDL compiler

For the purposes of this example, the PL/I source is generated in the first step of the supplied *`orbixhlq`*`.DEMOS.PLI.BLD.JCL(SIMPLIDL)` JCL. This JCL is used to run the Orbix IDL compiler for the simple persistent POA-based server demonstration supplied with your installation.

Example of the batch SIMPLIDL JCL

The following is the supplied JCL to run the Orbix IDL compiler for the batch version of the simple persistent POA-based server demonstration:

```
//SIMPLIDL JOB   (),
//          CLASS=A,
//          MSGCLASS=X,
//          MSGLEVEL=(1,1),
//          REGION=0M,
//          TIME=1440,
//          NOTIFY=&SYSUID,
//          COND=(4,LT)
//*----------------------------------------------------------------
//* Orbix -  Generate the PL/I files for the Simple Demo
//*----------------------------------------------------------------
//          JCLLIB ORDER=(orbixhlq.PROCS)
//          INCLUDE MEMBER=(ORXVARS)
//*
//* Make the following changes before running this JCL:
//*
//* 1.   Change 'SET DOMAIN='DEFAULT@' to your configuration
//*      domain name.
//*
//             SET DOMAIN='DEFAULT@'
//*
//IDLPLI   EXEC ORXIDL,
//          SOURCE=SIMPLE,
//          IDL=&ORBIX..DEMOS.IDL,
//          IDLPARM='-pli:-V'
//ITDOMAIN DD DSN=&ORBIX..CONFIG(&DOMAIN),DISP=SHR
```

The preceding JCL generates PL/I include members from an IDL member called `SIMPLE` (see the `SOURCE=SIMPLE` line).

Note: PL/I include members are always generated by default when you run the Orbix IDL compiler.

The preceding JCL specifies only the `-V` argument with the Orbix IDL compiler (see the `IDLPARM` line). This instructs the Orbix IDL compiler not to generate the *`idlmembername`*`V` server mainline source code member. See "Orbix IDL Compiler Arguments" on page 242 for more details.

Note: The preceding JCL is specific to the batch version of the supplied simple persistent POA-based server demonstration, and is contained in *`orbixhlq`*`.DEMOS.PLI.BLD.JCL(SIMPLIDL)`. For details of the JCL for the CICS or IMS version of the demonstration see "Example of the SIMPLIDL JCL" on page 59 or "Example of the SIMPLIDL JCL" on page 102.

Description of the JCL

The settings and data definitions contained in the preceding JCL can be explained as follows:

`ORBIX`	The high-level qualifier for your Orbix Mainframe installation, which is set in *`orbixhlq`*`.PROCS(ORXVARS)`.
`SOURCE`	The IDL member to be compiled.
`IDL`	The PDS for the IDL member.
`COPYLIB`	The PDS for the PL/I include members generated by the Orbix IDL compiler.
`IMPL`	The PDS for the PL/I source code members generated by the Orbix IDL compiler.
`IDLPARM`	The plug-in to the Orbix IDL compiler to be used (in the preceding example, it is the PL/I plug-in), and any arguments to be passed to it (in the preceding example, there is one argument specified, `-V`). See "Specifying Compiler Arguments" on page 244 for details of how to specify the Orbix IDL compiler arguments as parameters to it.

Running the Orbix IDL Compiler in UNIX System Services

Overview

This subsection describes how to run the Orbix IDL compiler in OS/390 UNIX System Services. It discusses the following topics:

- "Orbix IDL compiler configuration" on page 238.
- "Prerequisites to running the Orbix IDL compiler" on page 238.
- "Running the Orbix IDL compiler" on page 235.

Note: Even though you can run the Orbix IDL compiler in OS/390 UNIX System Services, Orbix does not support subsequent building of Orbix PL/I applications in OS/390 UNIX System Services.

Orbix IDL compiler configuration

The Orbix IDL compiler uses the Orbix IDL configuration file for its settings. The configuration file is set via the `IT_IDL_CONFIG_PATH` export variable.

Prerequisites to running the Orbix IDL compiler

Before you can run the Orbix IDL compiler, enter the following command to initialize your Orbix environment (where *`YOUR_ORBIX_INSTALL`* represents the full path to your Orbix installation directory):

```
cd $YOUR_ORBIX_INSTALL/etc/bin
. default-domain_env.sh
```

Note: You only need to do this once per logon.

Running the Orbix IDL compiler

The general format for running the Orbix IDL compiler is:

```
idl -pli[:-argument1][:-argument2][…] idlfilename.idl
```

In the preceding example, *`[:-argument1]`* and *`[:-argument2]`* represent optional arguments that can be passed as parameters to the Orbix IDL compiler, and *`idlfilename`* represents the name of the IDL file from which you want to generate the PL/I source and include files.

For example, consider the following command:

```
idl -pli:-V simple.idl
```

The preceding command instructs the Orbix IDL compiler to use the `simple.idl` file. The Orbix IDL compiler always generates PL/I include files by default, and the `-V` argument indicates that it should not generate an *`idlfilename`*`V` server mainline source code file. See "Orbix IDL Compiler Arguments" on page 242 for more details of Orbix IDL compiler arguments. See "Generated PL/I Source and Include Members" on page 240 and "Orbix IDL Compiler Configuration" on page 263 for more details of default generated filenames.

Generated PL/I Source and Include Members

Overview

This section describes the various PL/I source code and include members that the Orbix IDL compiler can generate.

Generated members

Table 21 provides an overview and description of the PL/I source code and include members that the Orbix IDL compiler can generate, based on the IDL member name.

Table 21: *Generated Source Code and Include Members*

Member Name	Member Type	Compiler Argument Used to Generate	Description
*idlmembername*I	Source code	-S	This is the server implementation source code member. It contains procedure definitions for all the callable operations. It is only generated if you use the -S argument.
*idlmembername*V	Source code	Generated by default	This is the server mainline source code member. It is generated by default unless you specify the -V argument to prevent generation of it.
*idlmembername*D	Include member	Generated by default	This is the select include member. It contains a select statement that determines the appropriate implementation function for the attribute or operation being called.
*idlmembername*L	Include member	Generated by default	This is the alignment include member. It contains procedures to perform the PL/I alignment calculations on behalf of the PL/I runtime.
*idlmembername*M	Include member	Generated by default	This is the main include member. It stores all the PL/I structures and declarations.

Table 21: *Generated Source Code and Include Members*

Member Name	Member Type	Compiler Argument Used to Generate	Description
`idlmembername`T	Include member	Generated by default	This is the typedef include member. It stores the based identifier information (that is, the PL/I structure definitions for which no storage is allocated).
`idlmembername`X	Include member	Generated by default	This is the runtime include member. It contains information for the PL/I runtime about the contents of each interface.

Member name restrictions

If the IDL member name exceeds six characters, the Orbix IDL compiler uses only the first six characters of that name when generating the source code and include member names. This allows space for appending a one-character suffix to each generated member name, while allowing it to adhere to the seven-character maximum size limit for PL/I external procedure names, which are based by default on the generated member names. Member names (and filenames on OS/390 UNIX System Services) are always generated in uppercase.

Filename extensions on OS/390 UNIX System Services

If you are running the Orbix IDL compiler in OS/390 UNIX System Services, it is recommended (but not mandatory) that you specify certain extensions for the generated filenames via the configuration variables in the Orbix IDL configuration file. The recommended extension for both the server implementation source code and server mainline source code filename is `.pli` and can be set via the `PLIModuleExtension` configuration variable. The recommended extension for all include filenames is `.inc` and can be set via the `PLIIncludeExtension` configuration variable.

Note: The settings for `PLIModuleExtension` and `PLIIncludeExtension` are left blank by default in the Orbix IDL configuration file. See "PL/I Configuration Variables" on page 264 for more details.

Orbix IDL Compiler Arguments

Overview

This section describes the various arguments that you can specify as parameters to the Orbix IDL compiler.

In this section

This section discusses the following topics:

Summary of the arguments

Overview

This subsection provides an introductory overview of the various Orbix IDL compiler arguments. Each argument is described in more detail further on in this section.

Summary

The Orbix IDL compiler arguments can be summarized as follows:

- `-D` Generate source code and include files into specified directories rather than the current working directory.

  Note: This is relevant to OS/390 UNIX System Services only.
- `-M` Set up an alternative mapping scheme for data names.
- `-O` Override default include member names with a different name.
- `-S` Generate server implementation source code.
- `-T` Indicate whether server code is for batch, IMS, or CICS.
- `-V` Do not generate the server mainline source code.
- `-W` Indicate whether generated source code uses `put` or `display` calls when issuing messages.

All these arguments are optional. This means that they do not have to be specified as parameters to the Orbix IDL compiler.

Specifying Compiler Arguments

Overview

This subsection describes how to specify the available arguments as parameters to the Orbix IDL compiler, both in batch and in OS/390 UNIX System Services. It discusses the following topics:

- "Specifying compiler arguments in batch" on page 244.
- "Specifying compiler arguments in UNIX System Services" on page 244.

Specifying compiler arguments in batch

To denote the arguments that you want to specify as parameters to the Orbix IDL compiler, you can use the DD name, `IDLPARM`, in the JCL that you use to run it. See "Running the Orbix IDL Compiler" on page 234 for an example of the supplied `SIMPLIDL` JCL that is used to run the Orbix IDL compiler for the simple persistent POA-based server demonstration.

The parameters for the `IDLPARM` entry in the JCL take the following format:

```
//  IDLPARM='-pli[:-M[option][membername]][:-Omembername]
//       [:-S][:-T[option]][:-V]'
```

Each argument that you specify must be preceded by a colon followed by a hyphen (that is, `:-`) , with no spaces between any characters or any arguments.

Note: In the `Pli` scope of the *`orbixhlq`*`.CONFIG(IDL)` configuration member, if you set the `IsDefault` variable to `YES`, you do not need to specify the `-pli` switch in the `IDLPARM` line of the JCL. See "Orbix IDL Compiler Configuration" on page 263 for more details.

Specifying compiler arguments in UNIX System Services

The parameters to the Orbix IDL compiler in OS/390 UNIX System Services take the following format:

```
-pli[:-D[option][dir]][:-M[option][membername]][:-Omembername]
    [:-S][:-T[option]][:-V]
```

Each argument that you specify must be preceded by a colon followed by a hyphen (that is, `:-`), with no spaces between any characters or any arguments.

Note: In the Pli scope of the Orbix IDL configuration file that is specified via the IT_IDL_CONFIG_PATH export variable, if you set the IsDefault variable to YES, you do not need to specify the -cobol switch as a parameter to the Orbix IDL compiler. See "Orbix IDL Compiler Configuration" on page 263 for more details.

-D Argument

Overview

By default, when you run the Orbix IDL compiler in OS/390 UNIX System Services, it generates source code and include files into the current working directory. You can use the `-D` argument with the Orbix IDL compiler to redirect some or all of the generated output into alternative directories.

> **Note:** The `-D` argument is relevant only if you are running the Orbix IDL compiler on OS/390 UNIX System Services. It is ignored if you specify it when running the Orbix IDL compiler on native OS/390.

Specifying the -D argument

The `-D` argument takes two components: a sub-argument that specifies the type of file to be redirected, and the directory path into which the file should be redirected. The three valid sub-arguments, and the file types they correspond to, are as follows:

`i`	Include files
`m`	IDL map files
`s`	Source code files

You must specify the directory path directly after the sub-argument. There must be no spaces between the argument, sub-argument, and directory path. For example, consider the following command that instructs the Orbix IDL compiler to generate PL/I files based on the IDL in `myfile.idl`, and to place generated include files in `/home/tom/pli/incl` and generated source code in `/home/tom/pli/src`:

```
idl -pli:-Di/home/tom/pli/incl:-Ds/home/tom/pli/src myfile.idl
```

Alternatively, consider the following command that instructs the Orbix IDL compiler to generate an IDL mapping file called `myfile.map`, based on the IDL in `myfile.idl`, and to place that mapping file in `/home/tom/pli/map`:

```
idl -pli:-Dm/home/tom/pli/map:-Mcreate0myfile.map myfile.idl
```

> **Note:** See the rest of this section for more details of how to generate source code and IDL mapping files.

-M Argument

Overview

PL/I data names generated by the Orbix IDL compiler are based on fully qualified IDL interface names by default (that is, *`IDLmodulename(s)_IDLinterfacename_IDLvariablename`*). You can use the `-M` argument with the Orbix IDL compiler to define your own alternative mapping scheme for data names. This is particularly useful if your PL/I data names are likely to exceed the 31-character restriction imposed by the PL/I compiler.

Example of data names generated by default

The example can be broken down as follows:

1. Consider the following IDL:

```
module Banks{
    module IrishBanks{
        interface SavingsBank{attribute short accountbal;};
        interface NationalBank{};
        interface DepositBank{};
    };
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the data names shown in Table 22 by default for the specified interfaces:

Table 22: *Example of Default Generated Data Names*

Interface Name	Generated Data Name
`SavingsBank`	`Banks_IrishBanks_SavingsBank`
`NationalBank`	`Banks_IrishBanks_NationalBank`
`DepositBank`	`Banks_IrishBanks_DepositBank`

By using the `-M` argument, you can replace the fully scoped names shown in Table 22 with alternative data names of your choosing.

Defining IDLMAP DD card in batch

If you are running the Orbix IDL compiler in batch, and you want to specify the `-M` argument as a parameter to it, you must define a DD card for `IDLMAP` in the JCL that you use to run the Orbix IDL compiler. This DD card specifies the PDS for the mapping members generated by the Orbix IDL compiler. (There is one mapping member generated for each IDL member.) For example, you might define the DD card as follows in the JCL (where *orbixhlq* represents the high-level qualifier for your Orbix Mainframe installation):

```
//IDLMAP DD DISP=SHR,DSN=orbixhlq.DEMOS.PLI.MAP
```

You can define a DD card for `IDLMAP` even if you do not specify the `-M` argument as a parameter to the Orbix IDL compiler. The DD card is simply ignored if the `-M` argument is not specified.

Steps to generate alternative names with the -M argument

The steps to generate alternative data name mappings with the `-M` argument are:

Step	Action
1	Run the Orbix IDL compiler with the `-Mcreate` argument, to generate the mapping member, complete with the fully qualified names and their alternative mappings.
2	Edit (if necessary) the generated mapping member, to change the alternative name mappings to the names you want to use.
3	Run the Orbix IDL compiler with the `-Mprocess` argument, to generate PL/I include members with the alternative data names.

Step 1—Generate the mapping member

First, you must run the Orbix IDL compiler with the `-Mcreate` argument, to generate the mapping member, which contains the fully qualified names and the alternative name mappings.

If you are running the Orbix IDL compiler in batch, the format of the command in the JCL used to run the compiler is as follows (where *X* represents the scope level, and `BANK` is the name of the mapping member you want to create):

```
IDLPARM='-pli:-McreateXBANK',
```

If you are running the Orbix IDL compiler in OS/390 UNIX System Services, the format of the command to run the compiler is as follows (where *X* represents the scope level, `bank.map` is the name of the mapping file you want to create, and `myfile.idl` is the name of the IDL file):

```
-pli:-McreateXbank.map myfile.idl
```

Note: The name of the mapping member can be up to six characters long. If you specify a name that is greater than six characters, the name is truncated to the first six characters. In the case of OS/390 UNIX System Services, you do not need to assign an extension of `.map` to the mapping filename; you can choose to use any extension or assign no extension at all.

Generating mapping files into alternative directories

If you are running the Orbix IDL compiler in OS/390 UNIX System Services, the mapping file is generated by default in the working directory. If you want to place the mapping file elsewhere, use the `-Dm` argument in conjunction with the `-Mcreate` argument. For example, the following command (where *X* represents the scope level) creates a `bank.map` file based on the `myfile.idl` file, and places it in the `/home/tom/pli/map` directory:

```
-pli:-Dm/home/tom/pli/map:-McreateXbank.map myfile.idl
```

See “-D Argument” on page 246 for more details about the `-D` argument.

Scoping levels with the -Mcreate command

As shown in the preceding few examples, you can specify a scope level with the `-Mcreate` command. This specifies the level of scoping to be involved in the generated data names in the mapping member. The possible scope levels are:

- `0` Map fully scoped IDL names to unscoped PL/I names (that is, to the IDL variable name only).
- `1` Map fully scoped IDL names to partially scoped PL/I names (that is, to *`IDLinterfacename_IDLvariablename`*). The scope operator, `/`, is replaced with an underscore, `_`.
- `2` Map fully scoped IDL names to fully scoped PL/I names (that is, to *`IDLmodulename(s)_IDLinterfacename_IDLvariablename`*). The scope operator, `/`, is replaced with an underscore, `_`.

The following provides an example of the various scoping levels. The example can be broken down as follows:

1. Consider the following IDL:

```
module Banks{
    module IrishBanks{
        interface SavingsBank{attribute short accountbal;};
        interface NationalBank{void deposit(in long
            amount);};
    };
};
```

2. Based on the preceding IDL example, a `-Mcreate0BANK` command produces the `BANK` mapping member contents shown in Table 23.

Table 23: *Example of Level-0-Scoped Generated Data Names*

Fully Scoped IDL Names	Generated Alternative Names
`Banks`	`Banks`
`Banks/IrishBanks`	`IrishBanks`
`Banks/IrishBanks/SavingsBank`	`SavingsBank`
`Banks/IrishBanks/SavingsBank/accountbal`	`accountbal`
`Banks/IrishBanks/NationalBank`	`NationalBank`
`Banks/IrishBanks/NationalBank/deposit`	`deposit`

Alternatively, based on the preceding IDL example, a `-Mcreate1BANK` command produces the `BANK` mapping member contents shown in Table 24.

Table 24: *Example of Level-1-Scoped Generated Data Names*

Fully Scoped IDL Names	Generated Alternative Names
`Banks`	`Banks`
`Banks/IrishBanks`	`IrishBanks`

Table 24: *Example of Level-1-Scoped Generated Data Names*

Fully Scoped IDL Names	Generated Alternative Names
`Banks/IrishBanks/SavingsBank`	`SavingsBank`
`Banks/IrishBanks/SavingsBank/accountbal`	`SavingsBank_accountbal`
`Banks/IrishBanks/NationalBank`	`NationalBank`
`Banks/IrishBanks/NationalBank/deposit`	`NationalBank_deposit`

Alternatively, based on the preceding IDL example, a `-Mcreate2BANK` command produces the `BANK` mapping member contents shown in Table 25.

Table 25: *Example of Level-2-Scoped Generated Data Names*

Fully Scoped IDL Names	Generated Alternative Names
`Banks`	`Banks`
`Banks/IrishBanks`	`Banks_IrishBanks`
`Banks/IrishBanks/SavingsBank`	`Banks_IrishBanks_SavingsBank`
`Banks/IrishBanks/SavingsBank/accountbal`	`Banks_IrishBanks_SavingsBank_accountbal`
`Banks/IrishBanks/NationalBank`	`Banks_IrishBanks_NationalBank`
`Banks/IrishBanks/NationalBank/deposit`	`Banks_IrishBanks_NationalBank_deposit`

Step 2—Change the alternative name mappings

You can manually edit the mapping member to change the alternative names to the names that you want to use. For example, you might change the mappings in the `BANK` mapping member as follows:

Table 26: *Example of Modified Mapping Names*

Fully Scoped IDL Names	Modified Names
`Banks/IrishBanks`	`IrishBanks`
`Banks/IrishBanks/SavingsBank`	`MyBank`
`Banks/IrishBanks/NationalBank`	`MyOtherBank`
`Banks/IrishBanks/SavingsBank/accountbal`	`Myaccountbalance`

Note the following rules:

- The fully scoped name and the alternative name meant to replace it must be separated by one space (and one space only).
- If the alternative name exceeds 31 characters, it is abbreviated to 31 characters, subject to the the normal PL/I mapping rules for identifiers.
- The fully scoped IDL names generated are case sensitive, so that they match the IDL being processed. If you add new entries to the mapping member, to cater for additions to the IDL, the names of the new entries must exactly match the corresponding IDL names in terms of case.

Step 3—Generate the PL/I include members

When you have changed the alternative mapping names as necessary, run the Orbix IDL compiler with the `-Mprocess` argument, to generate your PL/I include members complete with the alternative data names that you have set up in the specified mapping member.

If you are running the Orbix IDL compiler in batch, the format of the command to generate PL/I include members with the alternative data names is as follows (where `BANK` is the name of the mapping member you want to create):

```
IDLPARM='-pli:-MprocessBANK'
```

If you are running the Orbix IDL compiler in OS/390 UNIX System Services, the format of the command to generate PL/I include members with the alternative data names is as follows (where `bank.map` is the name of the mapping file you want to create):

```
-pli:-Mprocessbank.map
```

Note: If you are running the Orbix IDL compiler in OS/390 UNIX System Services, and you used the `-Dm` argument with the `-Mcreate` argument, so that the mapping file is not located in the current working directory, you must specify the path to that alternative directory with the `-Mprocess` argument. For example, `-pli:-Mprocess/home/tom/pli/map/bank.map`.

When you run the `-Mprocess` command, your PL/I include members are generated with the alternative data names you want to use, instead of with the fully qualified data names that the Orbix IDL compiler generates by default.

-O Argument

Overview

PL/I source code and include member names generated by the Orbix IDL compiler are based by default on the IDL member name. You can use the `-O` argument with the Orbix IDL compiler to map the default source and include member names to an alternative naming scheme, if you wish.

The `-O` argument is, for example, particularly useful for users who have migrated from IONA's Orbix 2.3-based solution for OS/390, and who want to avoid having to change the `%include` statements in their existing application source code. In this case, they can use the `-O` argument to automatically change the generated source and include member names to the alternative names they want to use.

Note: If you are an existing user who has migrated from IONA's Orbix 2.3-based solution for OS/390, see the *Mainframe Migration Guide* for more details.

Example of include members generated by Orbix IDL compiler

The example can be broken down as follows:

1. Consider the following IDL, where the IDL is stored in a member called TEST:

```
interface simple
{
     void sizeofgrid(in long mysize1, in long
         mysize2);
};

interface block
{
    void area(in long myarea);
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following PL/I include members, based on the IDL member name:
   - TESTD
   - TESTL
   - TESTM

- TESTT
- TESTX

Specifying the -O argument

If you are running the Orbix IDL compiler in batch, the following piece of JCL, for example, changes the include member names from `TEST` to `SIMPLE`:

```
//   SOURCE=TEST
//   …
//   IDLPARM='-pli:-OSIMPLE'
```

If you are running the Orbix IDL compiler in OS/390 UNIX System Services, the following command, for example, changes the include member names from `TEST` to `SIMPLE`:

```
-pli:-OSIMPLE test.idl
```

You must specify the alternative name directly after the `-O` argument (that is, no spaces). Even if you specify the replacement name in lower case (for example, `simple` instead of `SIMPLE`), the Orbix IDL compiler automatically generates replacement names in upper case.

Limitation in size of replacement name

If the name you supply as the replacement exceeds six characters (in the preceding example it does not, because it is `SIMPLE`), only the first six characters of that name are used as the basis for the alternative member names.

-S Argument

Overview

The -S argument generates skeleton server implementation source code (that is, the *idlmembername*I member). This member provides a skeleton implementation for the attributes and operation procedures to be implemented. It is not generated by default by the Orbix IDL compiler. It is only generated if you use the -S argument, because doing so overwrites any server implementation code that has already been created based on that IDL member name.

WARNING: Only specify the -S argument if you want to generate new server implementation source code or deliberately overwrite existing code.

Specifying the -S argument

If you are running the Orbix IDL compiler in batch, the following piece of JCL, for example, creates a server implementation member called SIMPLEI, based on the SIMPLE IDL member:

```
//   SOURCE=SIMPLE,
//   …
//   IDLPARM='-pli:-S'
```

If you are running the Orbix IDL compiler in OS/390 UNIX System Services, the following command, for example, creates a server implementation file called SIMPLEI, based on the simple.idl IDL file:

```
-pli:-S simple.idl
```

Note: In the case of OS/390 UNIX System Services, if you use the PLIModuleExtension configuration variable to specify an extension for the server implementation source code member name, this extension is automatically appended to the generated member name. The preceding commands generate batch server implementation code. If you want to generate CICS or IMS server implementation code, see "-T Argument" on page 257 for more details.

-T Argument

Overview

The `-T` argument allows you to specify whether the server code you want to generate is for use in batch, IMS, or CICS.

Qualifying parameters

The -T argument must be qualified by `NATIVE`, `IMS`, or `CICS`. For example:

`NATIVE`

Specifying `-TNATIVE` generates batch server mainline code. Specifying `-TNATIVE` with `-S` generates batch server implementation code.

Specifying `-TNATIVE` is the same as not specifying `-T` at all. That is, unless you specify `-TIMS`*x* or `TCICS`, the IDL compiler generates server code by default for use in native batch mode.

Note: If you specify `-TNATIVE` with `-V`, it prevents the generation of batch server mainline code.

`IMS`*x*

Specifying `-TIMS`*x* generates IMS server mainline code. Specifying `-TIMS`*x* with `-S` generates IMS server implementation code.

Specifying `-TIMS`*x* means that `io_pcb_ptr`, `alt_pcb_ptr`, and *x* number of extra pcb pointer parameters are added to the server mainline. It also means that the line `%include IMSPCB;` is added to the server mainline. Specifying `-TIMS` is the same as specifying `-TIMS0` (that is, if you do not specify a number, no extra pcb pointer parameters are added).

If you also specify the `-S` argument with the compiler, the line `%include IMSPCB;` is also added to the server implementation. IORs for the interfaces that server implements are not written to file, because the IMS adapter handles this.

Note: `IMSPCB` is a static include file that allows the server implementation to access the IMS pointers that are passed in the server mainline. If you specify `-TIMS`*x* with `-V`, it prevents the generation of IMS server mainline code.

`CICS`

Specifying `-TCICS` generates CICS server mainline code. Specifying `-TCICS` with `-S` generates CICS server implementation code.

Note: If you specify `-TCICS` with `-V`, it prevents the generation of CICS server mainline code.

Specifying the -TNATIVE argument

If you are running the Orbix IDL compiler in batch, the following piece of JCL, for example, creates a batch PL/I server mainline member (called TESTV) and a batch PL/I server implementation member (called TESTI), based on the TEST IDL member:

```
//    SOURCE=TEST,
//    …
//    IDLPARM='-pli:-S:-TNATIVE',
```

If you are running the Orbix IDL compiler in OS/390 UNIX System Services, the following command, for example, creates a batch PL/I server mainline file (called TESTV) and a batch PL/I server implementation file (called TESTI), based on the test.idl IDL file:

```
-pli:-S:-TNATIVE test.idl
```

Note: Specifying -TNATIVE is the same as not specifying -T at all.

See “Developing the Server” on page 28 for an example of batch PL/I server mainline and implementation members.

Specifying the -TIMS*x* argumentS

If you are running the Orbix IDL compiler in batch, the following piece of JCL, for example, creates an IMS PL/I server mainline member (called TESTV) with four PCB pointers, and an IMS PL/I server implementation member (called TESTI), based on the TEST IDL member:

```
//    SOURCE=TEST,
//    …
//    IDLPARM='-pli:-S:-TIMS4',
```

If you are running the Orbix IDL compiler in OS/390 UNIX System Services, the following command, for example, creates an IMS PL/I server mainline file (called TESTV) with four PCB pointers, and an IMS PL/I server implementation file (called TESTI), based on the test.idl IDL file:

```
-pli:-S:-TIMS4 test.idl
```

See “Developing the IMS Server” on page 67 for an example of IMS PL/I server mainline and implementation members.

Specifying the -TCICS argument

If you are running the Orbix IDL compiler in batch, the following piece of JCL, for example, creates a CICS PL/I server mainline member (called TESTV) and a CICS PL/I server implementation member (called TESTI), based on the TEST IDL member:

```
//    SOURCE=TEST,
//    …
//    IDLPARM='-pli:-S:-TCICS',
```

If you are running the Orbix IDL compiler in OS/390 UNIX System Services, the following command, for example, creates a CICS PL/I server mainline file (called TESTV) and a CICS PL/I server implementation file (called TESTI), based on the `test.idl` IDL file:

```
-pli:-S:-TCICS test.idl
```

See “Developing the CICS Server” on page 110 for an example of CICS PL/I server mainline and implementation members.

-V Argument

Overview

The -V argument prevents generation of server mainline source code (that is, it prevents generation of the *idlmembername*V member). You typically use this argument if you have added code that you do not want to be overwritten (for example, code that produces server output indicating that the server is ready to receive requests).

WARNING: If you do not specify the -V argument, any previous version of the server mainline source code member is overwritten.

Specifying the -V argument

If you are running the Orbix IDL compiler in batch, the following piece of JCL, for example, only generates include members, based on the SIMPLE IDL member, and prevents creation of a server mainline source code member called SIMPLEV:

```
//   SOURCE=SIMPLE,
//   …
//   IDLPARM='-pli:-V'
```

If you are running the Orbix IDL compiler in OS/390 UNIX System Services, the following command, for example, only generates include files, based on the simple.idl IDL file, and prevents creation of a server mainline source code file called SIMPLEV:

```
-pli:-V simple.idl
```

Note: In the case of OS/390 UNIX System Services, if you use the PLIModuleExtension configuration variable to specify an extension for the server mainline source code member name, this extension is automatically appended to the generated member name when you do not specify the -V argument. The preceding commands generate batch server implementation code. If you want to generate CICS or IMS server implementation code, see “-T Argument” on page 257 for more details.

-W Argument

Overview

The `-W` argument indicates whether the generated source code should use `put` or `display` calls when issuing messages.

Specifying the -W argument

The `-W` argument takes a sub-argument that specifies how to issue messages in the generated code. There are two valid sub-arguments:

`d` Use the PL/I `DISPLAY` call to issue messages in the generated code.

The `DISPLAY` call causes messages to be generated by a WTO.

A `DISPLAY` call issued from within an IMS region does not generate any messages. No messages are written to the system log or to `STDOUT`.

When using Enterprise PL/I, the `DISPLAY` compiler option can be used to specify where the messages are directed, as follows:

- `DISPLAY(WTO)` directs messages to the system log.
- `DISPLAY(STD)` directs messages to `STDOUT`.

`p` Use the PL/I `PUT` call to issue messages in the generated code. This is the default option.

The `PUT` call directs messages to `STDOUT`.

If you are running the Orbix IDL compiler in batch, the following JCL, for example, generates code that uses `DISPLAY` calls to issue messages:

```
//SOURCE=TEST
//…
//IDLPARM='-pli:-Wd'
```

If you are running the Orbix IDL compiler on OS/390 UNIX System Services, the following command, for example, generates code that uses DISPLAY calls to issue messages:

```
idl -pli:-Wd simple.idl
```

If you do not specify a `-W` argument, the default is to generate code that uses `PUT` calls to issue messages.

Orbix IDL Compiler Configuration

Overview

This section describes the configuration variables relevant to the Orbix IDL compiler `-pli` plug-in for PL/I source code and include member generation, and the `-mfa` plug-in for IMS or CICS adapter mapping member generation.

Note: The `-mfa` plug-in is not relevant for batch application development.

In this section

This section discusses the following topics:

PL/I Configuration Variables

Overview

The Orbix IDL configuration member contains settings for PL/I, along with settings for C++ and several other languages. If the Orbix IDL compiler is running in batch, it uses the configuration member located in *orbixhlq*.CONFIG(IDL). If the Orbix IDL compiler is running in OS/390 UNIX System Services, it uses the configuration file specified via the IT_IDL_CONFIG_PATH export variable.

Configuration variables

The PL/I configuration is listed under Pli as follows:

```
Pli
{
      Switch = "pli";
      ShlibName = "ORXBPLI";
      ShlibMajorVersion = "x";
      IsDefault = "NO";
      PresetOptions = "";

#     MainIncludeSuffix = "Q";

#     PL/I modules and includes extensions
#     The default is .pli and .inc on NT and none for OS/390.
      PLIModuleExtension = "";
      PLIIncludeExtension = "";
};
```

Note: Settings listed with a # are considered to be comments and are not in effect. The default in relation to PL/I modules and includes extensions is also none for OS/390 UNIX System Services.

Mandatory settings

The Switch, ShlibName, and ShlibMajorVersion variables are mandatory and their default settings must not be altered. They inform the Orbix IDL compiler how to recognize the PL/I switch, and what name the DLL plug-in is stored under. The *x* value for ShlibMajorVersion represents the version number of the supplied ShlibName DLL.

User-defined settings

All but the first three settings are user-defined and can be changed. The reason for these user-defined settings is to allow you to change, if you wish, default configuration values that are set during installation. To enable a user-defined setting, use the following format:

```
setting_name = "value";
```

List of available variables

Table 27 provides an overview and description of the available configuration variables.

Table 27: *Summary of PL/I Configuration Variables (Sheet 1 of 2)*

Variable Name	Description	Default
`IsDefault`	Indicates whether PL/I is the language that the Orbix IDL compiler generates by default from IDL. If this is set to `YES`, you do not need to specify the `-pli` switch when running the compiler.	
`PresetOptions`	The arguments that are passed by default as parameters to the Orbix IDL compiler.	
`PLIModuleExtension`[a]	Extension for the server source code filenames on OS/390 UNIX System Services or Windows NT. **Note:** This is left blank by default, and you can set it to any value you want. The recommended setting is `.pli`.	

Table 27: *Summary of PL/I Configuration Variables (Sheet 2 of 2)*

Variable Name	Description	Default
`PLIIncludeExtension`[a]	Extension for PL/I include filenames on OS/390 UNIX System Services or Windows NT. **Note:** This is left blank by default, and you can set it to any value you want. The recommended setting is `.inc`.	
`MainIncludeSuffix`	Suffix for the main include member name.	`M`
`TypedefIncludeSuffix`	Suffix for the typedef include member name.	`T`
`RuntimeIncludeSuffix`	Suffix for the runtime include member name.	`X`
`SelectIncludeSuffix`	Suffix for the select include member name.	`D`
`ServerMainModuleSuffix`	Suffix for the server mainline source code member name.	`V`
`ServerImplModuleSuffix`	Suffix for the server implementation source code member name.	`I`
`MaxFixedDigits`	Maximum precision for the `FIXED DECIMAL` type.	`15`
`NotSymbol`	Symbol for the `NOT` operator.	`¬`
`OrSymbol`	Symbol for the `OR` operator.	|[b]

a. This is ignored on native OS/390.

b. PL/I uses a double `OR` symbol (that is, `||`) as a string concatenation symbol.

The last nine variables shown in Table 27 on page 265 are not listed by default in *orbixhlq*.CONFIG(IDL). If you want to change the value for a variable name that is not listed by default, you must manually enter that variable name and its corresponding value in *orbixhlq*.CONFIG(IDL).

Note: Suffixes for member names can only be a single character. Use an asterisk (that is, *) if no suffix is to be used for a particular source code or include member.

Adapter Mapping Member Configuration Variables

Overview

The `-mfa` plug-in allows the Orbix IDL compiler to generate:

- IMS or CICS adapter mapping members from IDL, using the `-t` argument.
- Type information members, using the `-inf` argument.

The Orbix IDL configuration member contains configuration settings relating to the generation of IMS or CICS adapter mapping members and type information members.

Note: See the *IMS Adapter Administrator's Guide* or *CICS Adapter Administrator's Guide* for more details about adapter mapping members and type information members.

Configuration variables

The IMS or CICS adapter mapping member configuration is listed under `MFAMappings` as follows:

```
MFAMappings
{
      Switch = "mfa";
      ShlibName = "ORXBMFA";
      ShlibMajorVersion = "x";
      IsDefault = "NO";
      PresetOptions = "";

#     Mapping & Type Info file suffix and ext. may be overridden
#     The default mapping file suffix is A
#     The default mapping file ext. is .map and none for OS/390
#     The default type info file suffix is B
#     The default type info file ext. is .inf and none for OS/390
#     MFAMappingExtension   = "";
#     MFAMappingSuffix      = "";
#     TypeInfoFileExtension = "";
#     TypeInfoFileSuffix    = ""'
};
```

Mandatory settings

The `Switch`, `ShlibName`, and `ShlibMajorVersion` variables are mandatory and their settings must not be altered. They inform the Orbix IDL compiler how to recognize the adapter mapping member switch, and what name the DLL plug-in is stored under. The *x* value for `ShlibMajorVersion` represents the version number of the supplied `ShlibName` DLL.

User-defined settings

All but the first three settings are user-defined and can be changed. The reason for these user-defined settings is to allow you to change, if you wish, default configuration values that are set during installation. To enable a user-defined setting, use the following format.

```
setting_name = "value";
```

List of available variables

Table 28 provides an overview and description of the available configuration variables.

Table 28: *Adapter Mapping Member Configuration Variables*

Variable Name	Description	Default
`IsDefault`	Indicates whether the Orbix IDL compiler generates adapter mapping members by default from IDL. If this is set to `YES`, you do not need to specify the `-mfa` switch when running the compiler.	
`PresetOptions`	The arguments that are passed by default as parameters to the Orbix IDL compiler for the purposes of generating adapter mapping members.	
`MFAMappingExtension`[a]	Extension for the adapter mapping filename on OS/390 UNIX System Services or Windows NT.	`map`

Table 28: *Adapter Mapping Member Configuration Variables*

Variable Name	Description	Default
`MFAMappingSuffix`	Suffix for the adapter mapping member name. If you do not specify a value for this, it is generated with an `A` suffix by default.	`A`
`TypeInfoFileExtension`[a]	Extension for the type information filename on OS/390 UNIX System Services or Windows NT.	`inf`
`TypeInfoFileSuffix`	Suffix for the type information member name. If you do not specify a value for this, it is generated with a `B` suffix by default.	`B`

a. This is ignored on native OS/390.

Providing Arguments to the IDL Compiler

Overview

The Orbix IDL compiler configuration can be used to provide arguments to the IDL compiler. Normally, IDL compiler arguments are supplied to the `ORXIDL` procedure via the `IDLPARM` JCL symbolic, which comprises part of the JCL PARM. The JCL PARM has a 100-character limit imposed by the operating system. Large IDL compiler arguments, coupled with locale environment variables, tend to easily approach or exceed the 100-character limit. To help avoid problems with the 100-character limit, IDL compiler arguments can be provided via a data set containing IDL compiler configuration statements.

IDL compiler argument input to ORXIDL

The `ORXIDL` procedure accepts IDL compiler arguments from three sources:

- The *orbixhlq*`.CONFIG(IDL)` data set—This is the main Orbix IDL compiler configuration data set. See "PL/I Configuration Variables" on page 264 for an example of the `Pli` configuration scope. See "Adapter Mapping Member Configuration Variables" on page 268 for an example of the `MFAMappings` configuration scope. The `Pli` and `MFAMappings` configuration scopes used by the IDL compiler are in *orbixhlq*`.CONFIG(IDL)`. IDL compiler arguments are specified in the `PresetOptions` variable.
- The `IDLARGS` data set—This data set can extend or override what is defined in the main Orbix IDL compiler configuration data set. The `IDLARGS` data set defines a `PresetOptions` variable for each configuration scope. This variable overrides what is defined in the main Orbix IDL compiler configuration data set.
- The `IDLPARM` symbolic of the `ORXIDL` procedure—This is the usual source of IDL compiler arguments.

Because the IDLPARM symbolic is the usual source for IDL compiler arguments, it might lead to problems with the 100-character JCL PARM limit. Providing IDL compiler arguments in the IDLARGS data set can help to avoid problems with the 100-character limit. If the same IDL compiler arguments are supplied in more than one input source, the order of precedence is as follows:

- IDL compiler arguments specified in the IDLPARM symbolic take precedence over identical arguments specified in the IDLARGS data set and the main Orbix IDL compiler configuration data set.
- The PresetOptions variable in the IDLARGS data set overrides the PresetOptons variable in the main Orbix IDL compiler configuration data set. If a value is specified in the PresetOptons variable in the main Orbix IDL compiler configuration data set, it should be defined (along with any additional IDL compiler arguments) in the PresetOptions variable in the IDLARGS data set.

Using the IDLARGS data set

The IDLARGS data set can help when IDL compiles are failing due to the 100-character limit of the JCL PARM. Consider the following JCL:

```
//IDLPLI     EXEC ORXIDL,
//           SOURCE=BANKDEMO,
//           IDL=&ORBIX..DEMOS.IDL,
//           COPYLIB=&ORBIX..DEMOS.PLI.PLINCL,
//           IMPL=&ORBIX..DEMOS.PLI.SRC,
//           IDLPARM='-pli:-MprocessBANK:-OBANK'
```

In the preceding example, all the IDL compiler arguments are provided in the IDLPARM JCL symbolic, which is part of the JCL PARM. The JCL PARM can also be comprised of an environment variable that specifies locale information. Locale environment variables tend to be large and use up many of the 100 available characters in the JCL PARM. If the 100-character limit is exceeded, some of the data in the IDLPARM JCL symbolic can be moved to the IDLARGS data set to reclaim some of the JCL PARM space. The preceding example can be recoded as follows:

```
//IDLPLI      EXEC ORXIDL,
//            SOURCE=BANKDEMO,
//            IDL=&ORBIX..DEMOS.IDL,
//            COPYLIB=&ORBIX..DEMOS.PLI.PLINCL,
//            IMPL=&ORBIX..DEMOS.PLI.SRC,
//            IDLPARM='-pli'
//IDLARGS     DD *
Pli {PresetOptions = "-MprocessBANK:-OBANK";};
/*
```

The IDLPARM JCL symbolic retains the -pli switch. The rest of the IDLPARM data is now provided in the IDLARGS data set, freeing up 21 characters of JCL PARM space.

The IDLARGS data set contains IDL configuration file scopes. These are a reopening of the scopes defined in the main IDL configuration file. In the preceding example, the IDLPARM JCL symbolic contains a -pli switch. This instructs the IDL compiler to look in the Pli scope of the IDLARGS dataset for any IDL compiler arguments that might be defined in the PresetOptions variable. Based on the preceding example, it finds -MprocessBANK:-OBANK.

The IDLARGS data set must be coded according to the syntax rules for the main Orbix IDL compiler configuration data set. See "PL/I Configuration Variables" on page 264 for an example of the Pli configuration scope. See "Adapter Mapping Member Configuration Variables" on page 268 for an example of the MFAMappings configuration scope.

Note: A long entry can be continued by coding a backslash character (that is, \) in column 72, and starting the next line in column 1.

Defining multiple scopes in the IDLARGS data set

The IDLARGS data set can contain multiple scopes. Consider the following JCL that compiles IDL for a CICS server:

```
//IDLPLI      EXEC ORXIDL,
//            SOURCE=NSTSEQ,
//            IDL=&ORBIX..DEMOS.IDL,
//            COPYLIB=&ORBIX..DEMOS.CICS.PLI.PLINCL,
//            IMPL=&ORBIX..DEMOS.CICS.PLI.SRC,
//            IDLPARM='-pli:-TCICS -mfa:-tNSTSEQSV'
```

The `IDLPARM` JCL symbolic contains both a `-pli` and `-mfa` switch. The preceding example can be recoded as follows:

```
//IDLPLI     EXEC ORXIDL,
//           SOURCE=NSTSEQ,
//           IDL=&ORBIX..DEMOS.IDL,
//           COPYLIB=&ORBIX..DEMOS.CICS.PLI.PLINCL,
//           IMPL=&ORBIX..DEMOS.CICS.PLI.SRC,
//           IDLPARM='-pli -mfa'
//IDLARGS    DD *
Pli {PresetOptions = "-TCICS";};
MFAMappings {PresetOptions = "-tNSTSEQSV";};
/*
```

The `IDLPARM` JCL symbolic retains the `-pli` and `-mfa` IDL compiler switches. The IDL compiler looks for `-pli` switch arguments in the `Pli` scope, and for `-mfa` switch arguments in the `MFAMappings` scope.

Memory Handling

Memory handling must be performed when using dynamic structures such as unbounded strings, bounded and unbounded sequences, and anys. This chapter provides details of responsibility for the allocation and subsequent release of dynamic memory for these complex types at the various stages of an Orbix PL/I application. It first describes in detail the memory handling rules adopted by the PL/I runtime for operation parameters relating to different dynamic structures. It then provides a type-specific breakdown of the APIs that are used to allocate and release memory for these dynamic structures.

In this chapter

This chapter discusses the following topics:

Note: See "API Reference" on page 305 for full API details.

Operation Parameters

Overview

This section describes in detail the memory handling rules adopted by the PL/I runtime for operation parameters relating to different types of dynamic structures, such as unbounded strings, bounded and unbounded sequences, and any types. Memory handling must be performed when using these dynamic structures. It also describes memory issues arising from the raising of exceptions.

In this section

The following topics are discussed in this section:

Bounded Sequences and Memory Management

Overview for IN parameters

Table 29 provides a detailed outline of how memory is handled for bounded sequences that are used as `in` parameters.

Table 29: *Memory Handling for IN Bounded Sequences*

Client Application	Server Application
1. SEQINIT 2. write 3. PODEXEC—(send)	
	4. PODGET—(receive, allocate) 5. read 6. PODPUT—(free)
7. SEQFREE	

Summary of rules for IN parameters

The memory handling rules for a bounded sequence used as an `in` parameter can be summarized as follows, based on Table 29:

1. The client calls `SEQINIT` to initialize the sequence information block and allocate memory for it.
2. The client initializes the sequence elements.
3. The client calls `PODEXEC`, which causes the client-side PL/I runtime to marshal the values across the network.
4. The server calls `PODGET`, which causes the server-side PL/I runtime to receive the sequence and implicitly allocate memory for it.
5. The server obtains the sequence value from the operation parameter buffer.
6. The server calls `PODPUT`, which causes the server-side PL/I runtime to implicitly free the memory allocated by the call to `PODGET`.
7. The client calls `SEQFREE` to free the memory allocated by the call to `SEQINIT`.

Overview for INOUT parameters

Table 30 provides a detailed outline of how memory is handled for bounded sequences that are used as `inout` parameters.

Table 30: *Memory Handling for INOUT Bounded Sequences*

Client Application	Server Application
1. SEQINIT 2. write 3. PODEXEC—(send)	
	4. PODGET—(receive, allocate) 5. read 6. SEQFREE 7. SEQINIT 8. write 9. PODPUT—(send, free)
10. (free, receive, allocate) 11. read 12. SEQFREE	

Summary of rules for INOUT parameters

The memory handling rules for a bounded sequence used as an `inout` parameter can be summarized as follows, based on Table 30:

1. The client calls `SEQINIT` to initialize the sequence information block and allocate memory for it.
2. The client initializes the sequence elements.
3. The client calls `PODEXEC`, which causes the client-side PL/I runtime to marshal the values across the network.
4. The server calls `PODGET`, which causes the server-side PL/I runtime to receive the sequence and implicitly allocate memory for it.
5. The server obtains the sequence value from the operation parameter buffer.
6. The server calls `SEQFREE` to explicitly free the memory allocated for the original `in` sequence via the call to `PODGET` in point 4.
7. The server calls `SEQINIT` to initialize the replacement `out` sequence and allocate memory for it.
8. The server initializes the sequence elements for the replacement `out` sequence.

9. The server calls `PODPUT`, which causes the server-side PL/I runtime to marshal the replacement `out` sequence across the network and then implicitly free the memory allocated for it via the call to `SEQINIT` in point 7.
10. Control returns to the client, and the call to `PODEXEC` in point 3 now causes the client-side PL/I runtime to:
    i. Free the memory allocated for the original `in` sequence via the call to `SEQINIT` in point 1.
    ii. Receive the replacement `out` sequence.
    iii. Allocate memory for the replacement `out` sequence.

> **Note:** By having `PODEXEC` free the originally allocated memory before allocating the replacement memory means that a memory leak is avoided.

11. The client obtains the sequence value from the operation parameter buffer.
12. The client calls `SEQFREE` to free the memory allocated for the replacement `out` sequence via the call to `PODEXEC` in point 3.

Overview for OUT and return parameters

Table 31 provides a detailed outline of how memory is handled for bounded sequences that are used as `out` or `return` parameters.

Table 31: *Memory Handling for OUT and Return Bounded Sequences*

Client Application	Server Application
1. PODEXEC—(send)	
	2. PODGET—(receive) 3. SEQINIT 4. write 5. PODPUT—(send, free)
6. (receive, allocate) 7. read 8. SEQFREE	

Summary of rules for OUT and return parameters

The memory handling rules for a bounded sequence used as an `out` or `return` parameter can be summarized as follows, based on Table 31:

1. The client calls `PODEXEC`, which causes the client-side PL/I runtime to marshal the request across the network.
2. The server calls `PODGET`, which causes the server-side PL/I runtime to receive the client request.
3. The server calls `SEQINIT` to initialize the sequence and allocate memory for it.
4. The server initializes the sequence elements.
5. The server calls `PODPUT`, which causes the server-side PL/I runtime to marshal the values across the network and implicitly free the memory allocated to the sequence via the call to `SEQINIT`.
6. Control returns to the client, and the call to `PODEXEC` in point 1 now causes the client-side PL/I runtime to receive the sequence and implicitly allocate memory for it.
7. The client obtains the sequence value from the operation parameter buffer.
8. The client calls `SEQFREE`, which causes the client-side PL/I runtime to free the memory allocated for the sequence via the call to `PODEXEC`.

Unbounded Sequences and Memory Management

Overview for IN parameters

Table 32 provides a detailed outline of how memory is handled for unbounded sequences that are used as `in` parameters.

Table 32: *Memory Handling for IN Unbounded Sequences*

Client Application	Server Application
1. SEQALOC 2. SEQSET 3. PODEXEC—(send)	
	4. PODGET—(receive, allocate) 5. SEQGET 6. PODPUT—(free)
7. SEQFREE	

Summary of rules for IN parameters

The memory handling rules for an unbounded sequence used as an `in` parameter can be summarized as follows, based on Table 32:

1. The client calls `SEQALOC` to initialize the sequence information block and allocate memory for both the sequence information block and the sequence data.
2. The client calls `SEQSET` to initialize the sequence elements.
3. The client calls `PODEXEC`, which causes the client-side PL/I runtime to marshal the values across the network.
4. The server calls `PODGET`, which causes the server-side PL/I runtime to receive the sequence and implicitly allocate memory for it.
5. The server calls `SEQGET` to obtain the sequence value from the operation parameter buffer.
6. The server calls `PODPUT`, which causes the server-side PL/I runtime to implicitly free the memory allocated by the call to `PODGET`.
7. The client calls `SEQFREE` to free the memory allocated by the call to `SEQALOC`.

Overview for INOUT parameters

Table 33 provides a detailed outline of how memory is handled for unbounded sequences that are used as `inout` parameters.

Table 33: *Memory Handling for INOUT Unbounded Sequences*

Client Application	Server Application
1. SEQALOC 2. SEQSET 3. PODEXEC—(send)	
	4. PODGET—(receive, allocate) 5. SEQGET 6. SEQFREE 7. SEQALOC 8. SEQSET 9. PODPUT—(send, free)
10. (free, receive, allocate) 11. SEQGET 12. SEQFREE	

Summary of rules for INOUT parameters

The memory handling rules for an unbounded sequence used as an `inout` parameter can be summarized as follows, based on Table 33:

1. The client calls `SEQALOC` to initialize the sequence information block and allocate memory for both the sequence information block and the sequence data.
2. The client calls `SEQSET` to initialize the sequence elements.
3. The client calls `PODEXEC`, which causes the client-side PL/I runtime to marshal the values across the network.
4. The server calls `PODGET`, which causes the server-side PL/I runtime to receive the sequence and implicitly allocate memory for it.
5. The server calls `SEQGET` to obtain the sequence value from the operation parameter buffer.
6. The server calls `SEQFREE` to explicitly free the memory allocated for the original `in` sequence via the call to `PODGET` in point 4.
7. The server calls `SEQALOC` to initialize the replacement `out` sequence and allocate memory for both the sequence information block and the sequence data.

8. The server calls SEQSET to initialize the sequence elements for the replacement out sequence.
9. The server calls PODPUT, which causes the server-side PL/I runtime to marshal the replacement out sequence across the network and then implicitly free the memory allocated for it via the call to SEQALOC in point 7.
10. Control returns to the client, and the call to PODEXEC in point 3 now causes the client-side PL/I runtime to:
    i. Free the memory allocated for the original in sequence via the call to SEQALOC in point 1.
    ii. Receive the replacement out sequence.
    iii. Allocate memory for the replacement out sequence.

> **Note:** By having PODEXEC free the originally allocated memory before allocating the replacement memory means that a memory leak is avoided.

11. The client calls SEQGET to obtain the sequence value from the operation parameter buffer.
12. The client calls SEQFREE to free the memory allocated for the replacement out sequence in point 10 via the call to PODEXEC in point 3.

Overview for OUT and return parameters

Table 34 provides a detailed outline of how memory is handled for unbounded sequences that are used as out or return parameters.

Table 34: *Memory Handling for OUT and Return Unbounded Sequences*

Client Application	Server Application
1. PODEXEC—(send)	
	2. PODGET—(receive) 3. SEQALOC 4. SEQSET 5. PODPUT—(send, free)
6. (receive, allocate) 7. SEQGET 8. SEQFREE	

Summary of rules for OUT and return parameters

The memory handling rules for an unbounded sequence used as an `out` or `return` parameter can be summarized as follows, based on Table 34:

1. The client calls `PODEXEC`, which causes the client-side PL/I runtime to marshal the request across the network.
2. The server calls `PODGET`, which causes the server-side PL/I runtime to receive the client request.
3. The server calls `SEQALOC` to initialize the sequence and allocate memory for both the sequence information block and the sequence data.
4. The server calls `SEQSET` to initialize the sequence elements.
5. The server calls `PODPUT`, which causes the server-side PL/I runtime to marshal the values across the network and implicitly free the memory allocated to the sequence via the call to `SEQALOC`.
6. Control returns to the client, and the call to `PODEXEC` in point 1 now causes the client-side PL/I runtime to receive the sequence and implicitly allocate memory for it.
7. The client calls `SEQGET` to obtain the sequence value from the operation parameter buffer.
8. The client calls `SEQFREE`, which causes the client-side PL/I runtime to free the memory allocated for the sequence via the call to `PODEXEC`.

Unbounded Strings and Memory Management

Overview for IN parameters

Table 35 provides a detailed outline of how memory is handled for unbounded strings that are used as `in` parameters.

Table 35: *Memory Handling for IN Unbounded Strings*

Client Application	Server Application
1. STRSET 2. PODEXEC—(send)	
	3. PODGET—(receive, allocate) 4. STRGET 5. PODPUT—(free)
6. STRFREE	

Summary of rules for IN parameters

The memory handling rules for an unbounded string used as an `in` parameter can be summarized as follows, based on Table 35:

1. The client calls `STRSET` to initialize the unbounded string and allocate memory for it.
2. The client calls `PODEXEC`, which causes the client-side PL/I runtime to marshal the values across the network.
3. The server calls `PODGET`, which causes the server-side PL/I runtime to receive the string and implicitly allocate memory for it.
4. The server calls `STRGET` to obtain the string value from the operation parameter buffer.
5. The server calls `PODPUT`, which causes the server-side PL/I runtime to implicitly free the memory allocated by the call to `PODGET`.
6. The client calls `STRFREE` to free the memory allocated by the call to `STRSET`.

Overview for INOUT parameters

Table 36 provides a detailed outline of how memory is handled for unbounded strings that are used as `inout` parameters.

Table 36: *Memory Handling for INOUT Unbounded Strings*

Client Application	Server Application
1. STRSET 2. PODEXEC—(send)	
	3. PODGET—(receive, allocate) 4. STRGET 5. STRFREE 6. STRSET 7. PODPUT—(send, free)
8. (free, receive, allocate) 9. STRGET 10. STRFREE	

Summary of rules for INOUT parameters

The memory handling rules for an unbounded string used as an `inout` parameter can be summarized as follows, based on Table 36:

1. The client calls `STRSET` to initialize the unbounded string and allocate memory for it.
2. The client calls `PODEXEC`, which causes the client-side PL/I runtime to marshal the values across the network.
3. The server calls `PODGET`, which causes the server-side PL/I runtime to receive the string and implicitly allocate memory for it.
4. The server calls `STRGET` to obtain the string value from the operation parameter buffer.
5. The server calls `STRFREE` to explicitly free the memory allocated for the original `in` string via the call to `PODGET` in point 3.
6. The server calls `STRSET` to initialize the replacement `out` string and allocate memory for it.
7. The server calls `PODPUT`, which causes the server-side PL/I runtime to marshal the replacement `out` string across the network and then implicitly free the memory allocated for it via the call to `STRSET` in point 6.

8. Control returns to the client, and the call to PODEXEC in point 2 now causes the client-side PL/I runtime to:
   i. Free the memory allocated for the original in string via the call to STRSET in point 1.
   ii. Receive the replacement out string.
   iii. Allocate memory for the replacement out string.

   Note: By having PODEXEC free the originally allocated memory before allocating the replacement memory means that a memory leak is avoided.

9. The client calls STRGET to obtain the replacement out string value from the operation parameter buffer.
10. The client calls STRFREE to free the memory allocated for the replacement out string in point 8 via the call to PODEXEC in point 2.

Overview for OUT and return parameters

Table 37 provides a detailed outline of how memory is handled for unbounded strings that are used as out or return parameters.

Table 37: *Memory Handling for OUT and Return Unbounded Strings*

Client Application	Server Application
1. PODEXEC—(send)	
	2. PODGET—(receive) 3. STRSET 4. PODPUT—(send, free)
5. (receive, allocate) 6. STRGET 7. STRFREE	

Summary of rules for OUT and return parameters

The memory handling rules for an unbounded string used as an out or return parameter can be summarized as follows, based on Table 37:

1. The client calls PODEXEC, which causes the client-side PL/I runtime to marshal the request across the network.
2. The server calls PODGET, which causes the server-side PL/I runtime to receive the client request.

3. The server calls STRSET to initialize the string and allocate memory for it.
4. The server calls PODPUT, which causes the server-side PL/I runtime to marshal the values across the network and implicitly free the memory allocated to the string via the call to STRSET.
5. Control returns to the client, and the call to PODEXEC in point 1 now causes the client-side PL/I runtime to receive the string and implicitly allocate memory for it.
6. The client calls STRGET to obtain the string value from the operation parameter buffer.
7. The client calls STRFREE, which causes the client-side PL/I runtime to free the memory allocated for the string in point 5 via the call to PODEXEC in point 1.

Object References and Memory Management

Overview for IN parameters

Table 38 provides a detailed outline of how memory is handled for object references that are used as `in` parameters.

Table 38: *Memory Handling for IN Object References*

Client Application	Server Application
1. Attain object reference 2. PODEXEC—(send)	
	3. PODGET—(receive) 4. read 5. PODPUT
6. OBJREL	

Summary of rules for IN parameters

The memory handling rules for an object reference used as an `in` parameter can be summarized as follows, based on Table 38:

1. The client attains an object reference through some retrieval mechanism (for example, by calling `STR2OBJ` or `OBJRIR`).
2. The client calls `PODEXEC`, which causes the client-side PL/I runtime to marshal the object reference across the network.
3. The server calls `PODGET`, which causes the server-side PL/I runtime to receive the object reference.
4. The server can now invoke on the object reference.
5. The server calls `PODPUT`, which causes the server-side PL/I runtime to implicitly free any memory allocated by the call to `PODGET`.
6. The client calls `OBJREL` to release the object.

Overview for INOUT parameters

Table 39 provides a detailed outline of how memory is handled for object references that are used as `in` parameters.

Table 39: *Memory Handling for INOUT Object References*

Client Application	Server Application
1. Attain object reference 2. PODEXEC—(send)	
	3. PODGET—(receive) 4. read 5. OBJREL 6. Attain object reference 7. OBJDUPL 8. PODPUT—(send)
9. (receive) 10. read 11. OBJREL	

Summary of rules for INOUT parameters

The memory handling rules for an object reference used as an `inout` parameter can be summarized as follows, based on Table 39:

1. The client attains an object reference through some retrieval mechanism (for example, by calling `STR2OBJ` or `OBJRIR`).
2. The client calls `PODEXEC`, which causes the client-side PL/I runtime to marshal the object reference across the network.
3. The server calls `PODGET`, which causes the server-side PL/I runtime to receive the object reference.
4. The server can now invoke on the object reference.
5. The server calls `OBJREL` to release the original `in` object reference.
6. The server attains an object reference for the replacement `out` parameter through some retrieval mechanism (for example, by calling `STR2OBJ` or `OBJRIR`).
7. The server calls `OBJDUPL` to increment the object reference count and to prevent the call to `PODPUT` in point 8 from causing the replacement `out` object reference to be released.
8. The server calls `PODPUT`, which causes the server-side PL/I runtime to marshal the replacement `out` object reference across the network.

9. Control returns to the client, and the call to `PODEXEC` in point 2 now causes the client-side PL/I runtime to receive the replacement `out` object reference.
10. The client can now invoke on the replacement object reference.
11. The client calls `OBJREL` to release the object.

Overview for OUT and return parameters

Table 40 provides a detailed outline of how memory is handled for object references that are used as `out` or `return` parameters.

Table 40: *Memory Handling for OUT and Return Object References*

Client Application	Server Application
1. PODEXEC—(send)	
	2. PODGET—(receive) 3. Attain object reference 4. OBJDUPL 5. PODPUT—(send)
6. (receive) 7. read 8. OBJREL	

Summary of rules for OUT and return parameters

The memory handling rules for an object reference used as an `out` or `return` parameter can be summarized as follows, based on Table 40:

1. The client calls `PODEXEC`, which causes the client-side PL/I runtime to marshal the request across the network.
2. The server calls `PODGET`, which causes the server-side PL/I runtime to receive the client request.
3. The server attains an object reference through some retrieval mechanism (for example, by calling `STR2OBJ` or `OBJRIR`).
4. The server calls `OBJDUPL` to increment the object reference count and to prevent the call to `PODPUT` in point 5 from causing the object reference to be released.
5. The server calls `PODPUT`, which causes the server-side PL/I runtime to marshal the object reference across the network.
6. Control returns to the client, and the call to `PODEXEC` in point 1 now causes the client-side PL/I runtime to receive the object reference.

7. The client can now invoke on the object reference.
8. The client calls `OBJREL` to release the object.

The any Type and Memory Management

Overview for IN parameters

Table 41 provides a detailed outline of how memory is handled for an any type that is used as an `in` parameter.

Table 41: *Memory Handling for IN Any Types*

Client Application	Server Application
1. TYPESET 2. ANYSET 3. PODEXEC—(send) 8. ANYFREE	4. PODGET—(receive, allocate) 5. TYPEGET 6. ANYGET 7. PODPUT—(free)

Summary of rules for IN parameters

The memory handling rules for an object reference used as an `in` parameter can be summarized as follows, based on Table 41:

1. The client calls `TYPESET` to set the type of the any.
2. The client calls `ANYSET` to set the value of the any and allocate memory for it.
3. The client calls `PODEXEC`, which causes the client-side PL/I runtime to marshal the values across the network.
4. The server calls `PODGET`, which causes the server-side PL/I runtime to receive the any value and implicitly allocate memory for it.
5. The server calls `TYPEGET` to obtain the typecode of the any.
6. The client calls `ANYGET` to obtain the value of the any from the operation parameter buffer.
7. The server calls `PODPUT`, which causes the server-side PL/I runtime to implicitly free the memory allocated by the call to `PODGET`.
8. The client calls `ANYFREE` to free the memory allocated by the call to `ANYSET`.

Overview for INOUT parameters

Table 42 provides a detailed outline of how memory is handled for an any type that is used as an inout parameter.

Table 42: *Memory Handling for INOUT Any Types*

Client Application	Server Application
1. TYPESET 2. ANYSET 3. PODEXEC—(send)	
	4. PODGET—(receive, allocate) 5. TYPEGET 6. ANYGET 7. ANYFREE 8. TYPESET 9. ANYSET 10. PODPUT—(send, free)
11. (free, receive, allocate) 12. TYPEGET 13. ANYGET 14. ANYFREE	

Summary of rules for INOUT parameters

The memory handling rules for an object reference used as an inout parameter can be summarized as follows, based on Table 42:

1. The client calls TYPESET to set the type of the any.
2. The client calls ANYSET to set the value of the any and allocate memory for it.
3. The client calls PODEXEC, which causes the client-side PL/I runtime to marshal the values across the network.
4. The server calls PODGET, which causes the server-side PL/I runtime to receive the any value and implicitly allocate memory for it.
5. The server calls TYPEGET to obtain the typecode of the any.
6. The server calls ANYGET to obtain the value of the any from the operation parameter buffer.
7. The server calls ANYFREE to explicitly free the memory allocated for the original in value via the call to PODGET in point 4.
8. The server calls TYPESET to set the type of the replacement any.

9. The server calls `ANYSET` to set the value of the replacement any and allocate memory for it.
10. The server calls `PODPUT`, which causes the server-side PL/I runtime to marshal the replacement any value across the network and then implicitly free the memory allocated for it via the call to `ANYSET` in point 9.
11. Control returns to the client, and the call to `PODEXEC` in point 3 now causes the client-side PL/I runtime to:
    i. Free the memory allocated for the original any via the call to `ANYSET` in point 2.
    ii. Receive the replacement any.
    iii. Allocate memory for the replacement any.

> **Note:** By having `PODEXEC` free the originally allocated memory before allocating the replacement memory means that a memory leak is avoided.

12. The client calls `TYPEGET` to obtain the typecode of the replacement any.
13. The client calls `ANYGET` to obtain the value of the replacement any from the operation parameter buffer.
14. The client calls `ANYFREE` to free the memory allocated for the replacement `out` string in point 11 via the call to `PODEXEC` in point 3.

Overview for OUT and return parameters

Table 43 provides a detailed outline of how memory is handled for an any type that is used as an `inout` parameter.

Table 43: *Memory Handling for OUT and Return Any Types*

Client Application	Server Application
1. PODEXEC—(send)	
	2. PODGET—(receive) 3. TYPESET 4. ANYSET 5. PODPUT—(send, free)
6. (receive, allocate) 7. TYPEGET 8. ANYGET 9. ANYFREE	

Summary of rules for OUT and return parameters

The memory handling rules for an object reference used as an `out` or return parameter can be summarized as follows, based on Table 43:

1. The client calls `PODEXEC`, which causes the client-side PL/I runtime to marshal the request across the network.
2. The server calls `PODGET`, which causes the server-side PL/I runtime to receive the client request.
3. The server calls calls `TYPESET` to set the type of the any.
4. The server calls `ANYSET` to set the value of the any and allocate memory for it.
5. The server calls `PODPUT`, which causes the server-side PL/I runtime to marshal the values across the network and implicitly free the memory allocated to the any via the call to `ANYSET`.
6. Control returns to the client, and the call to `PODEXEC` in point 1 now causes the client-side PL/I runtime to receive the any and implicitly allocate memory for it.
7. The client calls `TYPEGET` to obtain the typecode of the any.
8. The client calls `ANYGET` to obtain the value of the replacement any from the operation parameter buffer.

9. The client calls `ANYFREE`, which causes the client-side PL/I runtime to free the memory allocated for the any in point 6 via the call to `PODEXEC` in point 1.

User Exceptions and Memory Management

Overview

Table 44 provides a detailed outline of how memory is handled for user exceptions.

Table 44: *Memory Handling for User Exceptions*

Client Application	Server Application
1. PODEXEC—(send)	
	2. PODGET—(receive, allocate)
	3. write
	4. PODERR
	5. (free)
6. *Free*	

Summary of rules

The memory handling rules for raised user exceptions can be summarized as follows, based on Table 44:

1. The client calls `PODEXEC`, which causes the PL/I runtime to marshal the client request across the network.
2. The server calls `PODGET`, which causes the server-side PL/I runtime to receive the client request and allocate memory for any arguments (if necessary).
3. The server initializes the user exception block with the information for the exception to be raised.
4. The server calls `PODERR`, to raise the user exception.
5. The server-side PL/I runtime automatically frees the memory allocated for the user exception in point 3.

> **Note:** The PL/I runtime does not, however, free the argument buffers for the user exception. To prevent a memory leak, it is up to the server program to explicitly free active argument structures, regardless of whether they have been allocated automatically by the PL/I runtime or allocated manually. This should be done before the server calls `PODERR`.

6. The client must explicitly free the exception ID in the user exception header, by calling STRFREE. It must also free any exception data mapping to dynamic structures (for example, if the user exception information block contains a sequence, this can be freed by calling SEQFREE).

Memory Management Routines

Overview

This section provides examples of PL/I routines for allocating and freeing memory for various types of dynamic structures. These routines are necessary when sending arguments across the wire or when using user-defined IDL types as variables within PL/I.

Unbounded strings

Use STRSET to allocate memory for unbounded strings, and STRFREE to subsequently free this memory. For example:

```
/* allocation */
dcl my_pli_string       char(15) init('Testing 123');
dcl my_corba_string     ptr;

call strset(my_pli_string, my_corba_string,
   length(my_pli_string));

/* deletion */
call strfree(my_corba_string);
```

Unbounded wide strings

Use WSTRSET to allocate memory for unbounded wide strings, and WSTRFRE to subsequently free this memory. For example:

```
/* allocation */
dcl my_corba_wstring      ptr;

call wstrset(my_pli_graphic, my_corba_wstring,
   my_pli_graphic_length);

/* deletion */
call wstrfre(my_corba_wstring);
```

Typecodes

As described in "IDL-to-PL/I Mapping" on page 177, typecodes are mapped to a pointer. They are handled in PL/I as unbounded strings and should contain a value corresponding to one of the typecode keys generated by the Orbix IDL compiler. For example:

```
/* allocation */
dcl my_typecode ptr;
call strset(my_typecode_ptr, my_complex_type,
   length(my_complex_type));

/* deletion */
call strfree(my_typecode_ptr);
```

Bounded sequences

Use `SEQINIT` to initialize a bounded sequence. This dynamically creates a sequence information block that is used internally to record state. Use `SEQFREE` to free this footprint before shutdown, to prevent memory leakage. For example:

```
/* allocation */
call seqinit(my_bseq_attr.result.result_seq, my_bseq_type,
   length(my_bseq_type));

/* deletion */
call seqfree(my_bseq_attr.result.result_seq);
```

`SEQFREE` deletes only the memory allocated via the calls to `SEQINIT` and `SEQALOC`. Therefore, you should always free the inner sequence element data first, and then the sequence itself. For example, when freeing a sequence of sequence of strings, follow this order:

1. Use `STRFREE` to free the data elements for the inner sequence.
2. Use `SEQFREE` to free the inner sequence.
3. Use `SEQFREE` to free the outer sequence.

Unbounded sequences

Use `SEQALOC` to initialize an unbounded sequence. This dynamically creates a sequence information block that is used internally to record state, and allocates the memory required for sequence elements. You can use `SEQSET` and `SEQGET` to access the sequence elements. You can also use `SEQSET` to resize the sequence if the maximum size of the sequence is not large enough to contain another sequence element. Use SEQFREE to free memory allocated via SEQALOC. For example:

```
/* allocation */
call seqaloc(my_useq_attr.result.result_seq, my_useq_max,
   my_useq_type, length(my_useq_type));

/* deletion */
call seqfree(my_useq_attr.result.result_seq);
```

Note: SEQFREE does not recursively free inner element data, so you should call inner element data before calling SEQFREE.

The any type

Use TYPESET to initialize the any information status block and allocate memory for it. Then use ANYSET to set the type of the any. Use ANYFREE to free memory allocated via TYPESET. This frees the flat structure created via TYPESET and any dynamic structures that are contained within it. For example:

```
dcl my_corba_any ptr;
dcl my_long             fixed bin(31) init(123);

/* allocation */
call typeset(my_corba_any ptr, CORBA_TYPE_LONG,
   length(CORBA_TYPE_LONG));
call anyset(my_corba_any ptr, addr(my_long));

/* deletion */
call anyfree(my_corba_any ptr);
```

Part 2

Programmer's Reference

In this part

This part contains the following chapters:

API Reference

This chapter summarizes the API functions that are defined for the Orbix PL/I runtime, in pseudo-code. It explains how to use each function, with an example of how to call it from PL/I.

In this chapter

This chapter discusses the following topics:

API Reference Summary

Introduction

This section provides a summary of the available API functions, in alphabetic order. See "API Reference Details" on page 312 for more details of each function.

Summary listing

```
ANYFREE(inout PTR any_pointer)
// Frees memory allocated to an any.

ANYGET(in PTR any_pointer,
       out PTR any_data_buffer)
// Extracts data out of an any.

ANYSET(inout PTR any_pointer,
       in PTR any_data_buffer)
// Inserts data into an any.

MEMALOC(out PTR memory_pointer,
        in FIXED BIN(31) memory_size)
// Allocates memory at runtime from the program heap.

MEMDBUG(in PTR memory_pointer,
        in FIXED BIN(15) memory_dump_size,
        in CHAR(*) text_string,
        in FIXED BIN(15) text_string_length)
// Output a formatted memory dump for the specified block of
// memory.

MEMFREE(in PTR memory_pointer)
// Frees the memory allocated at the address passed in.

OBJDUPL(in PTR object_reference,
        out PTR duplicate_obj_ref)
// Duplicates an object reference.

OBJGTID(in PTR object_reference,
        out CHAR(*) object_id,
        in FIXED BIN(31) object_id_length)
// Retrieves the object ID from an object reference.
```

```
OBJNEW(in CHAR(*) server_name,
       in CHAR(*) interface_name,
       in CHAR(*) object_id,
       out PTR object_reference)
// Creates a unique object reference.

OBJREL(in PTR object_reference)
// Releases an object reference.

OBJRIR(out PTR object_reference,
       in CHAR(*) desired_service)
// Returns an object reference to an object through which a
// service such as the Naming Service can be used.

OBJ2STR(in PTR object_reference,
        out CHAR(*) object_string)
// Retrieves the object ID from an IOR.

ORBARGS(in CHAR(*) argument_string,
        in FIXED BIN(31) argument_string_length,
        in CHAR(*) orb_name,
        in FIXED BIN(31) orb_name_length)
// Initializes a client or server connection to an ORB.

PODERR(in PTR user_exception_buffer)
// Allows a PL/I server to raise a user exception for an
// operation.

PODEXEC(in PTR object_reference,
        in CHAR(*) operation_name,
        inout PTR operation_buffer,
        inout PTR user_exception_buffer)
// Invokes an operation on the specified object.

PODGET(in PTR operation_buffer)
// Marshals in and inout arguments for an operation on the server
// side from an incoming request.

PODINFO(out PTR status_info_pointer)
// Retrieves address of the PL/I runtime status structure.

PODPUT(out PTR operation_buffer)
// Marshals return, out, and inout arguments for an operation on
// the server side from an incoming request.

PODREG(in PTR interface_description)
// Describes an IDL interface to the PL/I runtime
```

```
PODREQ(in PTR request_details)
// Provides current request information.

PODRUN
// Indicates the server is ready to accept requests.

PODSRVR(in CHAR(*) server_name,
        in FIXED BIN(31) server_name_length)
// Sets the server name for the current server process.

PODSTAT(in PTR status_buffer)
// Registers the status information block.

PODTIME(in FIXED BIN(15) timeout_type,
        in FIXED BIN(31) timeout_value)
// Used by clients for setting the call timeout.
// Used by servers for setting the event timeout.

PODVER(out CHAR(*) runtime_id_version,
       out CHAR(*) runtime_compile_time_date)
//Returns PL/I runtime compile-time information.

SEQALOC(out PTR sequence_control_data,
        in FIXED BIN(31) sequence_size,
        in CHAR(*) typecode_key,
        in FIXED BIN(31) typecode_key_length)
// Allocates memory for an unbounded sequence.

SEQDUPL(in PTR sequence_control_data,
        out PTR dupl_seq_control_data)
// Duplicates an unbounded sequence control block.

SEQFREE(in PTR sequence_control_data)
// Frees the memory allocated to an unbounded sequence.

SEQGET(in PTR sequence_control_data,
       in FIXED BIN(31) element_number,
       out PTR sequence_data)
// Retrieves the specified element from an unbounded sequence.

SEQINIT(out PTR sequence_control_data,
        in CHAR(*) typecode_key,
        in FIXED BIN(31) typecode_key_length)
// Initializes a bounded sequence
```

```
SEQLEN(in PTR sequence_control_data,
       out FIXED BIN(31) sequence_size)
// Retrieves the current length of the sequence

SEQLSET(in PTR sequence_control_data,
        in FIXED BIN(31) new_sequence_size)
// Changes the number of elements in the sequence

SEQMAX(in PTR sequence_control_data,
       out FIXED BIN(31) max_sequence_size)
// Returns the maximum set length of the sequence
SEQSET(in PTR sequence_control_data,
       in FIXED BIN(31) element_number,
       in PTR sequence_data)
// Places the specified data into the specified element of an
// unbounded sequence.

STRCON(inout PTR string_pointer,
       in PTR addon_string_pointer)
// Concatenates two unbounded strings.

STRDUPL(in PTR string_pointer,
        out PTR duplicate_string_pointer)
// Duplicates a given unbounded string

STRFREE(in PTR string_pointer)
// Frees the storage used by an unbounded string

STRGET(in PTR string_pointer,
       out CHAR(*) string,
       in FIXED BIN(31) string_length)
// Copies the contents of an unbounded string to a PL/I string

STRLENG(in PTR string_pointer,
        out FIXED BIN(31) string_length)
// Returns the actual length of an unbounded string

STRSET(out PTR string_pointer,
       in CHAR(*) string,
       in FIXED BIN(31) string_length)
// Creates an unbounded string from a CHAR(n) data item.

STRSETS(out PTR string_pointer,
        in CHAR(*) string,
        in FIXED BIN(31) string_length)
// Creates an unbounded string from a CHAR(n) data item
```

```
STR2OBJ(in PTR object_string,
        out PTR object_reference)
// Creates an object reference from an interoperable object
   reference (IOR).

TYPEGET(in PTR any_pointer,
        out CHAR(*) typecode_key,
        in FIXED BIN(31) typecode_key_length)
// Extracts the type name from an any.

TYPESET(in PTR any_pointer,
        in CHAR(*) typecode_key,
        in FIXED BIN(31) typecode_key_length)
// Sets the type name of an any

WSTRCON(inout PTR string_pointer,
        in PTR addon_string_pointer)
// Concatenates two unbounded wide strings.

WSTRDUP(in PTR string_pointer,
        out PTR duplicate_string_pointer)
// Duplicates a given unbounded wide string.

WSTFRE(in PTR string_pointer)
// Frees the storage used by an unbounded wide string.

WSTRGET(in PTR string_pointer,
        out GRAPHIC(*) string,
        in FIXED BIN(31) string_length)
// Copies the contents of an unbounded wide string to a PL/I
// graphic

WSTRLEN(in PTR string_pointer,
        out FIXED BIN(31) string_length)
/ Returns the number of characters held in the wide string
// (excluding trailing nulls).

WSTRSET(out PTR string_pointer,
        in CHAR(*) string,
        in FIXED BIN(31) string_length)
// Creates an unbounded wide string from a GRAPHIC(n) data item

WSTRSTS(out PTR string_pointer,
        in CHAR(*) string,
        in FIXED BIN(31) string_length)
// Creates an unbounded wide string from a GRAPHIC(n) data item
```

Auxiliary function

```
CHECK_ERRORS(in CHAR(*) function_name)
    RETURNS(FIXED BIN(31) error_number)
// Tests the completion status of the last PL/I runtime call.
```

API Reference Details

Introduction

This section provides details of each available API function, in alphabetic order.

In this section

The following topics are discussed in this section:

ANYFREE

Synopsis

```
ANYFREE(inout PTR any_pointer);
// Frees memory allocated to an any.
```

Usage

Common to clients and servers.

Description

The ANYFREE function releases the memory held by an any type that is being used to hold a value and its corresponding typecode. Do not try to use the any type after freeing its memory, because doing so might result in a runtime error.

When you call the ANYSET function, it allocates memory to store the actual value of the any. When you call the TYPESET function, it allocates memory to store the typecode associated with the value to be marshalled. When you subsequently call ANYFREE, it releases the memory that has been allocated via ANYSET and TYPESET.

Parameters

The parameter for ANYFREE can be described as follows:

`any_pointer`	This is an `inout` parameter that is a pointer to the address in memory where the any is stored.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface test {
    attribute any myany;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *idlmembername*T include member (where *idlmembername* represents the name of the IDL member that contains the IDL definitions):

```
dcl 1 test_myany_type based,
    3 result             ptr           init(sysnull());
```

Based on the preceding IDL, the Orbix IDL compiler also generates the following code, in the *idlmembername*M include member:

```
dcl 1 test_myany_attr aligned like test_myany_type;
```

3. The following is an example of how to use ANYFREE in association with the preceding code:

```
dcl short_value            fixed bin(15) init(0);

/* Retrieve the short value out of the any type           */
/* NB: We have determined the any type contained a CORBA  */
/*     short type through calling TYPEGET and testing its */
/*     result.                                            */
call anyget(test_myany_attr.result, addr(short_value));
put skip list('myany contains the value', short_value);

…

/* We are now finished using the any type, so free its    */
/* storage.                                               */

call anyfree(test_myany_attr.result);
```

See also

- "ANYSET" on page 319.
- "TYPESET" on page 412.
- "Memory Handling" on page 275.

ANYGET

Synopsis

```
ANYGET(in PTR any_pointer,
       out PTR any_data_buffer);
// Extracts data out of an any.
```

Usage

Common to clients and servers.

Description

The ANYGET function provides access to the buffer value that is contained in an any. You should check to see what type of data is contained in the any, and then ensure you supply a data buffer that is large enough to receive its contents. Before you call ANYGET you can use TYPEGET to extract the type of the data contained in the any.

Parameters

The parameters for ANYGET can be described as follows:

any_pointer	This is an inout parameter that is a pointer to the address in memory where the any is stored.
any_data_buffer	This is an out parameter that is used to store the value extracted from the any. The address of this buffer is passed to ANYGET.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface test {
    attribute any myany;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *idlmembername*T include member (where *idlmembername* represents the name of the IDL member that contains the IDL definitions):

```
dcl 1 test_myany_type based,
    3 result              ptr           init(sysnull());
```

Based on the preceding IDL, the Orbix IDL compiler also generates the following code, in the *idlmembername*M include member:

```
dcl 1 test_myany_attr aligned like test_myany_type;
```

3. The following is an example of how to use ANYGET in association with the preceding code:

```
dcl short_value            fixed bin(15) init(0);
dcl long_value             fixed bin(31) init(0);

/* Retrieve the typecode of the any, so we know how to    */
/* manipulate the data within it.                         */
call typeget(test_myany_attr, test_typecode,
             test_typecode_length);

select(test_typecode);
    when(CORBA_SHORT) do;
        /* Retrieve the short value out of the any. */
        call anyget(test_myany_attr.result,
                    addr(short_value));
        put skip list('myany contains the value',
            short_value);
    end;
    when(CORBA_LONG) do;
        /* Retrieve the long value out of the any.  */
        call anyget(test_myany_attr.result,
                    addr(long_value));
        put skip list('myany contains the value',
            long_value);
    end;
    …
end;

/* Now we are finished with the any, so free its storage  */
call anyfree(test_myany_attr.result);
```

See also "ANYSET" on page 319.

ANYSET

Synopsis

```
ANYSET(inout PTR any_pointer,
       in PTR any_data_buffer)
// Inserts data into an any.
```

Usage

Common to clients and servers.

Description

The ANYSET function copies the supplied data, which is placed in the data buffer by the application, into the any. ANYSET allocates the memory that is required to store the value of the any. You must call TYPESET before calling ANYSET, to set the typecode of the any. Ensure that this typecode matches the type of the data being copied to the any.

The address of the data_buffer is passed as an OUT parameter to ANYSET.

Parameters

The parameters for ANYSET can be described as follows:

any_pointer	This is an inout parameter that is a pointer to the address in memory where the any is stored.
any_data_buffer	This is an in parameter that contains the data to be copied to the any. The address of this buffer is passed to ANYSET.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface test {
    attribute any myany;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *idlmembername*T include member (where *idlmembername* represents the name of the IDL member that contains the IDL definitions):

```
dcl 1 test_myany_type based,
    3 result              ptr           init(sysnull());
```

Based on the preceding IDL, the Orbix IDL compiler also generates the following code, in the *idlmembername*M include member:

```
dcl 1 test_myany_attr aligned like test_myany_type;
```

3. The following is an example of how to use ANYSET in association with the preceding code:

```
dcl float_value            float dec(6) init(3.14159);

/* The basic CORBA typecodes are declared in the CORBA    */
/* include file. Complex types in the IDL are defined in   */
/* the T-suffixed include file generated for that IDL      */
/* file.                                                   */
test_typecode = CORBA_TYPE_FLOAT;

call typeset(test_myany_attr.result, test_typecode, 1);
call anyset(test_myany_attr.result, addr(float_value);
```

Exceptions

A CORBA::BAD_INV_ORDER::TYPESET_NOT_CALLED exception is raised if the typecode of the any has not been set via the TYPESET function.

See also

- "ANYGET" on page 317.
- "TYPESET" on page 412.

MEMALOC

Synopsis

```
MEMALOC(out PTR memory_pointer,
        in FIXED BIN(31) memory_size)
// Allocates memory at runtime from the program heap.
```

Usage

Common to clients and servers.

Description

The `MEMALOC` function allocates the specified number of bytes of memory from the program heap at runtime, and returns a pointer to the start of this memory block. MEMALOC is used to allocate space for dynamic structures.

Parameters

The parameters for `MEMALOC` can be described as follows:

`memory_pointer`	This is an `out` parameter that contains a pointer to the allocated memory block.
`memory_size`	This is an `in` parameter that specifies in bytes the amount of memory that is to be allocated.

Example

The following is an example of how to use `MEMALOC` in a client or server program:

```
dcl memory_block           ptr            init(sysnull());
dcl size_of_memory_req     fixed bin(31)  init(32);

/* Allocate a block of 32 bytes of memory */
call memaloc(memory_block, size_of_memory_req);
if check_errors('memaloc') ^= completion_status_yes then return;
```

Exceptions

A `CORBA::NO_MEMORY` exception is raised if there is not enough memory available to complete the request. In this case, the pointer will contain a null value.

See also

"MEMFREE" on page 324.

MEMDBUG

Synopsis

```
MEMDBUG(in PTR memory_pointer,
        in FIXED BIN(15) memory_dump_size,
        in CHAR(*) text_string,
        in FIXED BIN(15) text_string_length)
// Output a formatted memory dump for the specified block of
// memory.
```

Usage

Common to clients and servers.

Description

The MEMDBUG function allows you to output a specified formatted segment of memory and a text description. It is used for debugging purposes only.

Parameters

The parameters for MEMDBUG can be described as follows:

memory_pointer	This is an in parameter that contains a pointer to the allocated memory block.
memory_dump_size	This is an in parameter that specifies in bytes the amount of memory that is to be allocated for the memory dump.
text_string	This is an in parameter that contains the text string relating to the memory dump.
text_string_length	This is an in parameter that specifies the length of the text string.

Example

The example can be broken down as follows:

1. The following code displays the contents of a struct, called my_struct:

```
call memdbug(addr(my_struct),64,'Memory dump of MY_STRUCT',24);
```

2. The preceding call produces a result such as the following:

```
DEBUG DUMP - MEMORY DUMP OF MY_STRUCT
00x3a598(00000): 0000E3C5 E2E340D9 C5E2E4D3 E3E20000 '..TEST
   RESULTS.'
00x3a598(00010): 00E98572 009CB99A 0000FFFF 00004040
   '.ZeÊ..........'
00x3a598(00020): 00000000 E2E3C1E3 C9E2E3C9 C3E20000
   '..STATISTICS..'
00x3a598(00030): 000046A2 A3998995 8700FFFF 40404000
   '..ãstrln9..   '
```

MEMFREE

Synopsis

```
MEMFREE(in PTR memory_pointer)
// Frees the memory allocated at the address passed in.
```

Usage

Common to clients and servers.

Description

The `MEMFREE` function releases dynamically allocated memory, by means of a a pointer that was originally obtained by using `MEMALOC`. Do not try to use this pointer after freeing it, because doing so might result in a runtime error.

Parameters

The parameter for `MEMFREE` can be described as follows:

`memory_pointer`	This is an `in` parameter that contains a pointer to the allocated memory block.

Example

The following is an example of how to use `MEMFREE` in a client or server program:

```
dcl memory_block            ptr            init(sysnull());
dcl size_of_memory_req      fixed bin(31) init(32);

call memaloc(memory_block, size_of_memory_req);
if check_errors('memaloc') ^= completion_status_yes then return;

…

/* Finished using the block of memory, so free it */
call memfree(memory_block);
```

See also

"MEMALOC" on page 321.

OBJDUPL

Synopsis

```
OBJDUPL(in PTR object_reference,
        out PTR duplicate_obj_ref)
// Duplicates an object reference.
```

Usage

Common to clients and servers.

Description

The `OBJDUPL` function creates a duplicate reference to an object. It returns a new reference to the original object reference and increments the reference count of the object. It is equivalent to calling `CORBA::Object::_duplicate()` in C++. Because object references are opaque and ORB-dependent, your application cannot allocate storage for them. Therefore, if more than one copy of an object reference is required, you can use `OBJDUPL` to create a duplicate.

Parameters

The parameters for `OBJDUPL` can be described as follows:

`object_reference`	This is an `in` parameter containing the valid object reference.
`duplicate_obj_ref`	This is an `out` parameter containing the duplicate object reference.

Example

The following code shows how `OBJDUPL` can be used within a server:

```
dcl 1 get_an_object_args,
    3 result                        ptr init(sysnull());

dcl test_prg_object                 ptr init(sysnull());
dcl my_object                       ptr init(sysnull());

…

/* test_prg_object already set up from earlier processing */
call podexec(test_prg_object,
             get_an_object,
             get_an_object_args,
             no_user_exceptions);
if check_errors('objdupl') ^= completion_status_yes then return;

/* Duplicate the returned object */
call objdupl(get_an_object_args.result,my_object);
if check_errors('objdupl') ^= completion_status_yes then return;

/* Processing done with the duplicated object reference */
…

/* Finished using the duplicated object reference, so free it */
call objrel(my_object);
if check_errors('objrel') ^= completion_status_yes then return;
```

See also

"OBJREL" on page 331 and "Object References and Memory Management" on page 289.

OBJGTID

Synopsis

```
OBJGTID(in PTR object_reference,
        out CHAR(*) object_id,
        in FIXED BIN(31) object_id_length)
// Retrieves the object ID from an object reference.
```

Usage

Specific to batch servers. Not relevant to CICS or IMS.

Description

The `OBJGTID` function retrieves the object ID string from an object reference. It is equivalent to calling `POA::reference_to_id` in C++.

Parameters

The parameters for `OBJGTID` can be described as follows:

`object_reference`	This is an `in` parameter that contains the valid object reference.
`object_id`	This is an `out` parameter that is a bounded string containing the object name relating to the specified object reference. If this string is not large enough to contain the object name, the returned string is truncated.
`object_id_length`	This is an `in` parameter that specifies the length of the object name.

Example

The following code shows how OBJGTID can be used within a client:

```
dcl object_id                    char(256);
dcl simple_obj                   ptr;

…

/* IOR is read from the file written by the server */
%include READIOR;

…

/* Create an object reference from the IOR */
call str2obj(iorrec_ptr,simple_obj);
if check_errors('str2obj')^=completion_status_yes then return;

/* Retrieve the object ID from the object reference */
call objgtid(simple_obj,object_id,length(object_id));
if check_errors('objgtid')^=completion_status_yes then return;

put skip list('Object ID retrieved: ' || object_id);
```

Exceptions

A CORBA::BAD_PARAM::LENGTH_TOO_SMALL exception is raised if the length of the string containing the object name is greater than the object_id_length parameter.

A CORBA::BAD_PARAM::INVALID_OBJECT_ID exception is raised if an Orbix 2.3 object reference is passed.

A CORBA::BAD_INV_ORDER::SERVER_NAME_NOT_SET exception is raised if PODSRVR is not called.

OBJNEW

Synopsis

```
OBJNEW(in CHAR(*) server_name,
       in CHAR(*) interface_name,
       in CHAR(*) object_id,
       out PTR object_reference)
// Creates a unique object reference.
```

Usage

Server-specific.

Description

The `OBJNEW` function creates a unique object reference that encapsulates the specified object identifier and interface names. The resulting reference can be returned to clients to initiate requests on that object. It is equivalent to calling `POA::create_reference_with_id` in C++.

Parameters

The parameters for `OBJNEW` can be described as follows:

`server_name`	This is an `in` parameter that is a bounded string containing the server name. This must be the same as the value passed to `PODSRVR`. This string must be terminated by at least one space.
`interface_name`	This is an `in` parameter that is a bounded string containing the interface name. This string must be terminated by at least one space. The *`idlmembername`*`T` include member contains a PL/I declaration for each interface defined in the relevant IDL member. These definitions are stored in the Interface List section and have a `_intf` suffix.
`object_id`	This is an `in` parameter that is a bounded string containing the object identifier name relating to the specified object reference. This string must be terminated by at least one space.
`object_reference`	This is an out parameter that contains the created object reference.

Example

The following is an example of how `OBJNEW` is typically used in a server program (where IOR variable declarations have been omitted for the sake of brevity):

```
dcl server_name         char(06)  init('SIMPLE ');
dcl interface_name      char(18)  init
    ('IDL:Simple/SimpleObject:1.0 ');
dcl my_object_id        char(10)  init('Simple_01 ');
dcl my_object           ptr       init(sysnull());

…
/* Register our interface with the PL/I runtime */
call podreg(simple_interface);

/* Now create an object reference for the server, so we  */
/* can use it to create an IOR, allowing clients to      */
/* invoke operations on our server.                      */
call objnew(server_name, interface_name, my_object_id,
            my_object);
if check_errors('objnew') ^= completion_status_yes then return;

/* Create the IOR  */
call obj2str(my_object, iorrec_ptr);
if check_errors(’obj2str’) ^= completion_status_yes then return;

/* Retrieve the string from the unbounded string  */
call strget(iorrec_ptr, iorrec, iorrec_len);
if check_errors(’strget’) ^= completion_status_yes then return;

/* Now we can write out our server IOR string to a file  */
write file(IORFILE) from(iorrec);
```

Exceptions

A `CORBA::BAD_PARAM::INVALID_SERVER_NAME` exception is raised if the server name does not match the server name passed to `ORBSRVR`.

A `CORBA::BAD_PARAM::NO_OBJECT_IDENTIFIER` exception is raised if the parameter for the object identifier name is an invalid string.

A `CORBA::BAD_INV_ORDER::INTERFACE_NOT_REGISTERED` exception is raised if the specified interface has not been registered via `ORBREG`.

A `CORBA::BAD_INV_ORDER::SERVER_NAME_NOT_SET` exception is raised if `PODSRVR` is not called.

OBJREL

Synopsis

```
OBJREL(in PTR object_reference)
// Releases an object reference.
```

Usage

Common to clients and servers.

Description

The `OBJREL` function indicates that the caller will no longer access the object reference. It is equivalent to calling `CORBA::release()` in C++. `OBJREL` decrements the reference count of the object reference.

Parameters

The parameter for `OBJREL` can be described as follows:

`object_reference`	This is an `in` parameter that contains the valid object reference.

Example

The following is an example of how OBJREL is typically used in a server program:

```
dcl 1 get_an_object_args,
      3 result            ptr init(sysnull());

dcl test_prg_object       ptr init(sysnull());
dcl my_object             ptr init(sysnull());

…

/* test_prg_object already set up from earlier processing */
call podexec(test_prg_object,
             get_an_object,
             get_an_object_args,
             no_user_exceptions);
if check_errors('objdupl') ^= completion_status_yes then return;

/* Duplicate the returned object */
call objdupl(get_an_object_args.result,my_object);
if check_errors('objdupl') ^= completion_status_yes then return;

/* Processing done with the duplicated object reference */
…

/* Finished using the duplicated object reference, so free it */
call objrel(my_object);
if check_errors('objrel') ^= completion_status_yes then return;
```

See also

"OBJDUPL" on page 325 and "Object References and Memory Management" on page 289.

OBJRIR

Synopsis

```
OBJRIR(out PTR object_reference,
       in CHAR(*) desired_service)
// Returns an object reference to an object through which a
// service such as the Naming Service can be used.
```

Usage

Common to batch clients and servers. Not relevant to CICS or IMS.

Description

The `OBJRIR` function returns an object reference, through which a service (for example, the Interface Repository or a CORBAservice like the Naming Service) can be used. For example, the Naming Service is accessed by using a `desired_service` string with the "NameService " value. It is equivalent to calling `ORB::resolve_initial_references()` in C++.

Table 45 shows the common services available, along with the PL/I identifier assigned to each service. The PL/I identifiers are declared in the CORBA include member.

Table 45: *Summary of Common Services and Their PL/I Identifiers*

Service	PL/I Identifier
InterfaceRepository	`IFR_SERVICE`
NameService	`NAMING_SERVICE`
TradingService	`TRADING_SERVICE`

Parameters

The parameters for `OBJRIR` can be described as follows:

`object_reference`	This is an `out` parameter that contains an object reference for the desired service.
`desired_service`	This is an `in` parameter that is a string specifying the desired service. This string is terminated by a space.

Exceptions

A `CORBA::ORB::InvalidName` exception is raised if the `desired_service` string is invalid.

Example

The following is an example of how to use `OBJRIR` in a client program, to obtain the object reference to the NameService (which is then used to retrieve the object reference for a server called `Simple`):

```
dcl name_service_obj       ptr  init(sysnull());
dcl simple_obj             ptr  init(sysnull());

/* Retrieve the object reference for the NameService */
call objrir(name_service_obj,naming_service);
if check_errors('objrir') ^= completion_status_yes then return;

/* The setting up of the resolve request to retrieve the  */
/* object reference for the Simple server is omitted here */
/* for brevity.                                           */
…

/* Call resolve on the NameService using the */
/* object reference retrieved via OBJRIR.   */
call podexec(name_service_obj,
             NamingContext_resolve,
             NamingContext_resolve_args,
             NAMING_user_exceptions);
if check_errors('podexec') ^= completion_status_yes then return;

/* Assign our simple_obj to the object reference */
/* retrieved from the call to the NameService.   */
simple_obj=NamingContext_resolve_args.result;

/* Now we have retrieved the object reference for our  */
/* client, we can invoke calls on it.                  */
/* Our example call below does not take any parameters */
/* so no setup is required prior to invoking.          */
call podexec(simple_obj,
             simple_call_me,
             addr(simple_call_me_args),
             no_user_exceptions);
if check_errors('podexec') ^= completion_status_yes then return;
…
```

OBJ2STR

Synopsis

```
OBJ2STR(in PTR object_reference,
        out CHAR(*) object_string)
// Retrieves the object ID from an IOR.
```

Usage

Common to batch clients and servers. Not relevant to CICS or IMS.

Description

The `OBJ2STR` function creates an interoperable object reference (IOR) from a valid object reference. The object reference string that is passed to `OBJ2STR` must be terminated with a null character. You can use the `STRSET` function to create this string.

Parameters

The parameters for `OBJ2STR` can be described as follows:

`object_reference`	This is an `in` parameter that contains the object reference.
`object_string`	This is an `out` parameter that contains the stringified representation of the object reference (that is, the IOR).

Example

The following example shows part of the server mainline code, generated in the *idlmembername*SV member by the Orbix IDL compiler, with added comments for clarity:

```
call objnew(server_name,
            Simple_SimpleObject_intf,
            Simple_SimpleObject_objid,
            Simple_SimpleObject_obj);
if check_errors('objnew') ^= completion_status_yes then return;

/* Write out the IOR for each interface                      */
open file(IORFILE);

call obj2str(Simple_SimpleObject_obj,
             iorrec_ptr);
if check_errors('obj2str') ^= completion_status_yes then return;

call strget(iorrect_ptr,iorrec,iorrec_len);
if check_errors('strget') ^= completion_status_yes then return;

write file(IORFILE) FROM(iorrec);
close file(IORFILE);
```

See also

"STR2OBJ" on page 404.

ORBARGS

Synopsis

```
ORBARGS(in CHAR(*) argument_string,
        in FIXED BIN(31) argument_string_length,
        in CHAR(*) orb_name,
        in FIXED BIN(31) orb_name_length)
// Initializes a client or server connection to an ORB.
```

Usage

Common to clients and servers.

Description

The `ORBARGS` function initializes a client or server connection to the ORB. It is equivalent to calling `CORBA::ORB_init()` in C++. It first initializes an application in the ORB environment and then it returns the ORB pseudo-object reference to the application for use in future ORB calls.

Because applications do not initially have an object on which to invoke ORB calls, `ORB_init()` is a bootstrap call into the CORBA environment. Therefore, the `ORB_init()` call is part of the `CORBA` module but is not part of the `CORBA::ORB` class.

The `arg_list` is optional and is usually not set. The use of the `orb_name` is recommended, because if it is not specified, a default ORB name is used.

Special ORB identifiers (indicated by either the `orb_name` parameter or the `-ORBid` argument) are intended to uniquely identify each ORB used within the same location domain in a multi-ORB application. The ORB identifiers are allocated by the ORB administrator who is responsible for ensuring that the names are unambiguous.

When you are assigning ORB identifiers via `ORBARGS`, if the `orb_name` parameter has a value, any `-ORBid` arguments in the `argv` are ignored. However, all other ORB arguments in `argv` might be significant during the ORB initialization process. If the `orb_name` parameter is null, the ORB identifier is obtained from the `-ORBid` argument of `argv`. If the `orb_name` is null and there is no `-ORBid` argument in `argv`, the default ORB is returned in the call.

Note: Orbix PL/I batch does not support the passing of arguments via PPARM at runtime. However, if you want to pass an ORB name at runtime, you can use a `DD:ORBARGS` instead.

Parameters

The parameters for ORBARGS can be described as follows:

argument_string	This is an in parameter that is a bounded string containing the argument list of the environment-specific data for the call. See "ORB arguments" for more details.
argument_string_length	This is an in parameter that specifies the length of the argument string list.
orb_name	This is an in parameter that is a bounded string containing the ORB identifier for the initialized ORB, which must be unique for each server across a location domain. However, client-side ORBs and other "transient" ORBs do not register with the locator, so it does not matter what name they are assigned.
orb_name_length	This is an in parameter that specifies the length of the ORB identifier string.

ORB arguments

Each ORB argument is a sequence of configuration strings or options of the following form:

```
-ORBsuffix value
```

The suffix is the name of the ORB option being set. The value is the value to which the option is set. There must be a space between the suffix and the value. Any string in the argument list that is not in one of these formats is ignored by the ORB_init() method.

Valid ORB arguments include:

-ORBboot_domain *value*	This indicates where to get boot configuration information.
-ORBdomain *value*	This indicates where to get the ORB actual configuration information.
-ORBid *value*	This is the ORB identifier.
-ORBname *value*	This is the ORB name.

Example

The following is an example of client code at ORB setup time:

```
dcl arg_list                    char(40)       init('');
dcl arg_list_len                fixed bin(31)  init(0);
dcl orb_name                    char(07)       init('simple ');
dcl orb_name_len                fixed bin(31)  init(6);

%include CORBA;
%include CHKERRS;
%include SIMPLEM;
%include SIMPLEX;
%include SETUPCL;               /* Various DCLs for the client */
%include IORFILE;               /* Describes the IOR File type */

open file(IORFILE) input;
%include READIOR;               /* Read in the server's IOR    */

/* Initialize the runtime status information block for */
alloc pod_status_information set(pod_status_ptr);
call podstat(pod_status_ptr);

/* Initialize the ORB connection with the name 'simple' */
call orbargs(arg_list, arg_list_len, orb_name, orb_name_len);
if check_errors('orbargs') ^= completion_status_yes then return;

/* Register the interface with the PL/I runtime */
call podreg(addr(Simple_SimpleObject_interface));
if check_errors('podreg') ^= completion_status_yes then return;
…
```

Note: The `%include CHKERRS` statement in the preceding example is used in server and batch client programs. It is replaced with `%include CHKCLCIC` in CICS client programs, and `%include CHKCLIMS` in IMS client programs.

Exceptions

A `CORBA::BAD_INV_ORDER::ADAPTER_ALREADY_INITIALIZED` exception is raised if `ORBARGS` is called more than once in a client or server.

PODERR

Synopsis

```
PODERR(in PTR user_exception_buffer)
// Allows a PL/I server to raise a user exception for an
// operation.
```

Usage

Server-specific.

Description

The PODERR function allows a PL/I server to raise a user exception for the operation that supports the exception(s), which can then be picked up on the client side via the user exception buffer that is passed to PODEXEC for the relevant operation. To raise a user exception, the server program must set the exception_id, the d discriminator, and the appropriate exception buffer.

The server calls PODERR instead of PODPUT in this instance, and this informs the client that a user exception has been raised. See "Memory Handling" on page 275 for more details. Calling PODERR does not terminate the server program.

The client can determine if a user exception has been raised, by testing to see whether the exception_id of the operation's user_exception_buffer passed to PODEXEC is equal to zero after the call. See "PODEXEC" on page 345 for an example of how a PL/I client determines if a user exception has been raised.

Parameters

The parameter for PODERR can be described as follows:

user_exception_buffer	This is an in parameter that contains the PL/I representation of the user exceptions that the IDL operations support. The address of the user exception buffer is passed to PODERR.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface test {
    exception bad {
        long          value;
        string<32>    reason;
    };

    exception critical {
        short          value_x;
        string<31>     likely_cause;
        string<63>     action_required;
    };

    long myop(in long number) raises(bad, critical);
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code for the user exception block, in the *idlmembername*M include member (where *idlmembername* represents the name of the IDL member that contains the IDL definitions):

```
/*------------------------------------------------------*/
/* Defined User Exceptions                              */
/*------------------------------------------------------*/
dcl 1 TEST_user_exceptions,
      3 exception_id           ptr,
      3 d                      fixed bin(31)  init(0),
      3 u                      ptr;

dcl 1 test_bad_exc_d           fixed bin(31)  init(1);
dcl 1 test_critical_exc_d      fixed bin(31)  init(2);

dcl 1 test_bad_exc            based(TEST_user_exceptions.u),
      3 idl_value              fixed bin(31)  init(0),
      3 reason                 char(32)       init('');

dcl 1 test_critical_exc
   based(TEST_user_exceptions.u),
      3 value_x                fixed bin(15)  init(0),
      3 likely_cause           char(31)       init(''),
      3 action_required        char(63)       init('');

dcl TEST_user_exceptions_area  area(96);
TEST_user_exceptions.u = addr(TEST_user_exceptions_area);
```

The following operation structure declaration is also generated in the *idlmembername*M include member:

```
dcl 1 test_myop_args aligned like test_myop_type;
```

The body of the operation structure is generated as follows, in the *idlmembername*T include member:

```
dcl 1 test_myop_type based,
    3 number                fixed bin(31)  init(0),
    3 result                fixed bin(31)  init(0);
```

3. The following piece of client code shows how the client calls PODERR:

```
test_myop_args.number = 42;
call podexec(test_obj, test_myop, addr(test_myop_args),
             addr(TEST_user_exceptions));
```

   Because the myop operation can throw user exceptions, the address of the user exception structure is passed as the fourth parameter.

4. The following piece of server code shows how the server can set up and throw an exception in the myop operation:

```
if myop_args.number = 0 then
    do;
        /* Set the exception ID */
        strset(TEST_user_exceptions.exception_id,
            test_bad_exid, test_bad_len);

        /* Set the exception discriminator */
        TEST_user_exceptions.d = test_bad_exc_d;
        test_bad_exc.idl_value = 9999;
       test_bad_exc.reason = 'Input must be greater than 0';
        call poderr(TEST_user_exceptions);
    end;
else
    do;
        …
```

5. A test such as the following can be set up in the client code to check for a user exception:

```
select(TEST_user_exceptions.d);
  when(no_exceptions_thrown) /* no user exception has */
                             /* been thrown           */
    put skip list('No exceptions thrown, return value is:',
      test_myop_args.result);
  when(test_bad_exc_d) do;
    put skip list('User exception ''bad'' was thrown:');
    put skip list('value returned was',
      test_bad_exc.idl_value);
    put skip list('reason returned was ' ||
      test_bad_exc.reason);
  end;
  when(test_critical_exc_d) do;
    put skip list('User exception ''critical'' was
      thrown:');
    put skip list('value_x returned was',
      test_critical_exc.value_x);
    put skip list('likely_cause was ' ||
      test_critical_exc.likely_cause);
    put skip list('action_required is ' ||
      test_critical_exc.action_required);
  end;
end;
```

Exceptions

The appropriate CORBA exception is raised if an attempt is made to raise a user exception that is not related to the invoked operation.

A `CORBA::BAD_PARAM::UNKNOWN_TYPECODE` exception is raised if the typecode cannot be determined when marshalling an `any` type or a user exception.

See also

- "PODEXEC" on page 345.
- The `BANK` demonstration in *`orbixhlq`*`.DEMOS.PLI.SRC` for a complete example of how to use `PODERR`.

PODEXEC

Synopsis

```
PODEXEC(in PTR object_reference,
        in CHAR(*) operation_name,
        inout PTR operation_buffer,
        inout PTR user_exception_buffer)
// Invokes an operation on the specified object.
```

Usage

Client-specific.

Description

The `PODEXEC` function allows a PL/I client to invoke operations on the server interface represented by the supplied object reference. All `in` and `inout` parameters must be set up prior to the call. `PODEXEC` invokes the specified operation for the specified object, and marshals and populates the operation buffer, depending on whether they are `in`, `out`, `inout`, or `return` arguments.

As shown in the following example, the client can test for a user exception by examining the `exception_id` of the operation's user `exception_buffer` after calling `PODEXEC`. A non-zero value indicates a user exception. A zero value indicates that no user exception was raised by the operation that the call to `PODEXEC` invoked. If an exception is raised, you must reset the discriminator of the user exception block to zero by setting the `discrim_d` to `no_user_exceptions_thrown`.

The following example is based on the `grid` demonstration. Some of the referenced data items in the example are found in the `GRIDM` and `GRIDX` include members. The address of the `operation_buffer` is passed to `PODEXEC` in the third argument.

Parameters

The parameters for `PODEXEC` can be described as follows:

`object_reference`	This is an `in` parameter that contains the valid object reference. You can use `STR2OBJ` to create this object reference.

`operation_name`	This is an `in` parameter that is a string containing the operation name to be invoked. This string is terminated by a space. It is defined in the *`idlmembername`*`M` and *`idlmembername`*`T` include members generated by the Orbix IDL compiler.
`operation_buffer`	This is an `inout` parameter that contains a PL/I structure of the data types that the operation supports. The address of the buffer is passed to `PODEXEC`. It is defined in the *`idlmembername`*`M` and *`idlmembername`*`T` include members generated by the Orbix IDL compiler.
`user_exception_buffer`	This is an `inout` parameter that contains the PL/I representation of the user exceptions that the IDL operations support. The address of the user exception buffer is passed to `PODEXEC`. It is defined in the *`idlmembername`*`M` and *`idlmembername`*`T` include members generated by the Orbix IDL compiler. If the operation can throw a user exception, the address of the associated user exception block is passed as this parameter. Where a user exception has not been defined, the `NO_USER_EXCEPTIONS` null pointer variable, which is defined in the `CORBA` include member, is used instead.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface test {
    string<32> call_me(in string<32> input_string);
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *`idlmembername`*`T` include member (where *`idlmembername`* represents the name of the IDL member that contains the IDL definitions):

```
dcl 1 test_call_me_type based,
    3 input_string             char(32)            init(''),
    3 result                   char(32)            init('');
```

Based on the preceding IDL, the Orbix IDL compiler also generates the following code, in the *idlmembername*M include member:

```
dcl 1 test_call_me_args aligned like test_call_me_type;
```

3. The following piece of client code shows how to call the `call_me` operation:

```
/* Register the test interface with the PL/I runtime */
call podreg(addr(test_interface));
if check_errors('podreg') ^= completion_status_yes then return;
/* Create an object reference from the server's IOR */
call str2obj(iorrec_ptr,test_obj);
if check_errors('objset')^=completion_status_yes then return;

/* Set up the input arguments */
test_call_me_args.input_string = 'hello';

/* We are now ready to call operation call_me */
call podexec(test_obj, test_call_me,
             addr(test_call_me_args), no_user_exceptions);
if check_errors('podexec') ^= completion_status_yes then return;

put skip list('result received back from call_me: ' ||
   test_call_me_args.result);
```

Exceptions

A `CORBA::BAD_INV_ORDER::INTERFACE_NOT_REGISTERED` exception is raised if the client tries to invoke an operation on an interface that has not been registered via `ORBREG`.

A `CORBA::BAD_PARAM::INVALID_DISCRIMINATOR_TYPECODE` exception is raised if the discriminator typecode is invalid when marshalling a union type.

A `CORBA::BAD_PARAM::UNKNOWN_OPERATION` exception is raised if the operation is not valid for the interface.

A `CORBA::BAD_PARAM::UNKNOWN_TYPECODE` exception is raised if the typecode cannot be determined when marshalling an any type or a user exception.

See also

The `BANK` demonstration in *orbixhlq*`.DEMOS.PLI.SRC` for a complete example of how to use `PODEXEC`.

PODGET

Synopsis

```
PODGET(in PTR operation_buffer)
// Marshals in and inout arguments for an operation on the server
// side from an incoming request.
```

Usage

Server-specific.

Description

Each operation implementation must begin with a call to `PODGET` and end with a call to `PODPUT`. Even if the operation takes no parameters and has no return value, you must still call `PODGET` and `PODPUT` and, in such cases, pass a dummy `CHAR(1)` data item, which the Orbix IDL compiler generates for such cases.

`PODGET` copies the incoming operation's argument values into the complete PL/I operation parameter buffer that is supplied. This buffer is generated automatically by the Orbix IDL compiler. Only `IN` and `INOUT` values in this structure are populated by this call.

The Orbix IDL compiler generates the call for `PODGET` in the *`idlmembername`*`D` include member, for each attribute and operation defined in the IDL.

Parameters

The parameter for `PODGET` can be described as follows:

`operation_buffer`	This is an `in` parameter that contains a PL/I structure representing the data types that the operation supports. The address of the buffer is passed to `PODGET`.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface foo {
    long bar(in short n, out short m);
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following structure definition in the *idlmembername*T include member (where *idlmembername* represents the name of the IDL member that contains the IDL definitions):

```
dcl 1 foo_bar_type based,
    3 n                     fixed bin(15)           init(0),
    3 m                     fixed bin(15)           init(0),
    3 result                fixed bin(31)           init(0);
```

3. The declaration in the *idlmembername*M include member is as follows:

```
dcl 1 foo_bar_args aligned like foo_bar_type;
```

4. A subset of the *idlmembername*D include member is as follows, with comments added for clarity:

```
select(interface);
  when(foo_tc) do;
    select(operation);
      when (foo_bar) do;
        /* Fill the foo_bar_args structure with the incoming  */
        /* data. The IN value 'n' will be filled.             */
        call podget(addr(foo_bar_args));
        if check_errors('podget') ^= completion_status_yes then
            return;

        /* Now call the user implementation code for op       */
        /* foo_bar.                                           */
        call proc_foo_bar(addr(foo_bar_args));

        /* Transmit the out value 'm' and result of op        */
        /* foo_bar.                                           */
        call podput(addr(foo_bar_args));
        if check_errors('podput') ^= completion_status_yes then
            return;
      end;
    otherwise;
    …
```

Exceptions

A `CORBA::BAD_INV_ORDER::ARGS_ALREADY_READ` exception is raised if the `in` or `inout` parameter for the request has already been processed.

A CORBA::BAD_PARAM::INVALID_DISCRIMINATOR_TYPECODE exception is raised if the discriminator typecode is invalid when marshalling a union type.

A CORBA::BAD_PARAM::UNKNOWN_TYPECODE exception is raised if the typecode cannot be determined when marshalling an any type or a user exception.

See also "PODPUT" on page 353.

PODINFO

Synopsis

```
PODINFO(out PTR status_info_pointer)
// Retrieves address of the PL/I runtime status structure.
```

Usage

Common to clients and servers.

Description

The PODINFO function obtains the address of pod_status_information. If the buffer has not been allocated, it is assigned a null value. Assuming that the buffer has been allocated elsewhere, and that it was followed subsequently by a call to PODSTAT, the call to PODINFO acts as if a call to PODSTAT has been made. This is because PODINFO recalls the address of the status_information_buffer through the pod_status_ptr (when it is used as shown in the following example). PODINFO allows the same status buffer to be used across multiple PL/I modules, which will be linked together later when the application is compiled.

Parameters

The parameter for PODINFO can be described as follows::

status_info_pointer	This is an out parameter that contains the address of the PL/I runtime status information structure.

Example

The following shows how pod_status_information is set up in the PL/I server mainline code, which the Orbix IDL compiler generates in the *idlmembername*V module:

```
alloc pod_status_information set(pod_status_ptr);
call podstat(pod_status_ptr);
```

The check_errors function uses pod_status_information to determine whether an error has occurred in the most recently called runtime function. However, because the check_errors function can be included from any PL/I module, and not just from the server mainline, you must call `PODINFO` to connect the `pod_status_information` buffer with the original buffer, via the `pod_status_ptr`. This is shown in the following piece of code from `check_errors`, with added comments for clarity:

```
/* pod_status_information is based on pod_status_ptr     */
/* podinfo retrieves the address of the block of memory  */
/* it was originally assigned to in the server program.  */
call podinfo(pod_status_ptr);

/* Now we have a link to the original status buffer      */
exception_number = pod_status_information.exception_number;

if exception_number = 0 then
    …
```

See also "PODSTAT" on page 364.

PODPUT

Synopsis

```
PODPUT(out PTR operation_buffer)
// Marshals return, out, and inout arguments for an operation on
// the server side from an incoming request.
```

Usage

Server-specific.

Description

Each operation implementation must begin with a call to `PODGET` and end with a call to `PODPUT`. The `PODPUT` function copies the operation's outgoing argument values from the complete PL/I operation parameter buffer passed to it. This buffer is generated automatically by the Orbix IDL compiler. Only `inout`, `out`, and the `result out` item are populated by this call.

You must ensure that all `inout`, `out`, and `result` values are correctly allocated (for dynamic types) and populated. If a user exception has been raised before calling `PODPUT`, no `inout`, `out`, or `result` parameters are marshalled, and nothing is returned in such cases. If a user exception has been raised, `PODERR` must be called instead of `PODPUT`, and no `inout`, `out`, or `result` parameters are marshalled. See "PODERR" on page 340 for more details.

The Orbix IDL compiler generates the call for `PODPUT` in the *`idlmembername`*`D` include member for each attribute and operation defined in the IDL.

Parameters

The parameter for `PODPUT` can be described as follows:

`operation_buffer`	This is an `out` parameter that contains a PL/I structure of the data types that the operation supports. The address of the buffer is passed to `PODPUT`.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface foo {
    long bar(in short n, out short m);
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following structure definition in the *idlmembername*T include member (where *idlmembername* represents the name of the IDL member that contains the IDL definitions):

```
dcl 1 foo_bar_type based,
    3 n                    fixed bin(15)           init(0),
    3 m                    fixed bin(15)           init(0),
    3 result               fixed bin(31)           init(0);
```

3. The declaration in the *idlmembername*M include member is as follows:

```
dcl 1 foo_bar_args aligned like foo_bar_type;
```

4. A subset of the *idlmembername*D include member is as follows, with comments added for clarity:

```
select(interface);
  when(foo_tc) do;
    select(operation);
      when (foo_bar) do;
        /* Fill the foo_bar_args structure with the incoming  */
        /* data. The IN value 'n' will be filled.             */
        call podget(addr(foo_bar_args));
        if check_errors('podget') ^= completion_status_yes then
          return;

        /* Now call the user implementation code for op       */
        /* foo_bar.                                           */
        call proc_foo_bar(addr(foo_bar_args));

        /* Transmit the out value 'm' and result of op        */
        /* foo_bar.                                           */
        call podput(addr(foo_bar_args));
        if check_errors('podput') ^= completion_status_yes then
          return;
      end;
    otherwise;
    …
```

Exceptions

A `CORBA::BAD_INV_ORDER::ARGS_NOT_READ` exception is raised if the `in` or `inout` parameters for the request have not been processed.

A CORBA::BAD_PARAM::INVALID_DISCRIMINATOR_TYPECODE exception is raised if the discriminator typecode is invalid when marshalling a union type.

A CORBA::BAD_PARAM::UNKNOWN_TYPECODE exception is raised if the typecode cannot be determined when marshalling an any type or a user exception.

See also "PODGET" on page 348.

PODREG

Synopsis

```
PODREG(in PTR interface_description)
// Describes an IDL interface to the PL/I runtime
```

Usage

Common to clients and servers.

Description

The PODREG function registers an interface with the PL/I runtime, by using the interface description that is stored in the *idlmembername*X include member, which the Orbix IDL compiler generates.

The Orbix IDL compiler generates an *idlmembername*X include member for each IDL interface (where *idlmembername* represents the name of the IDL member that contains the IDL definitions). The *idlmembername*X contains a structure for each interface, which the Orbix IDL compiler populates with information about the IDL. The PODREG function uses this populated interface information to register an interface with the PL/I runtime, for use in subsequent calls to PODGET and PODPUT.

You must call PODREG for every interface that the client or server uses. In this case, you must pass the address of the structure stored in the *idlmembername*X include member for each interface, to register the information about the interface with the PL/I runtime. The format for this structure name is *interface_name*_interface.

Parameters

The parameter for PODREG can be described as follows:

interface_description	This is an in parameter that contains the address of the interface definition.

Example

The following code shows part of the setup for a typical PL/I client:

```
/* Location of the interface descriptor(s) for the */
/* IDL file MYIDL, containing interface MyIntf     */
%include MYIDLX;
…

/* The server's IOR is read in, code omitted for brevity */
…

/* Initialize the client's connection to the ORB */
call orbargs(arg_list,
             arg_list_len,
             orb_name,
             orb_name_len);

/* Register interface MyIntf with the PL/I runtime */
call podreg(addr(MyIntf_interface));
if check_errors('podreg') ^= completion_status_yes then return;

/* Create an object reference from the IOR */
call str2obj(iorrec_ptr,shlong_obj);
if check_errors('str2obj')^=completion_status_yes then return;

/* Client is now ready to start setting up calls to the server */
```

Exceptions

A `CORBA::BAD_INV_ORDER::INTERFACE_ALREADY_REGISTERED` exception is raised if the client or server attempts to register the same interface more than once.

PODREQ

Synopsis

```
PODREQ(in PTR request_details)
// Provides current request information.
```

Usage

Server-specific.

Description

The server implementation module calls PODREQ to extract the relevant information about the current request. PODREQ provides information about the current invocation request in a request information buffer, which is defined as follows in the supplied DISPINIT include member:

```
dcl 1 reqinfo,
    3 interface_name       ptr;
    3 operation_name       ptr;
    3 principal            ptr;
    3 target               ptr;
```

In the preceding structure, the first three data items are unbounded CORBA character strings. You can use the STRGET function to copy the first three data items into CHAR(n) buffers. The TARGET item is the PL/I object reference for the operation invocation. After PODREQ is called, the structure contains the following data:

INTERFACE_NAME	The name of the interface, which is stored as an unbounded string.
OPERATION_NAME	The name of the operation for the invocation request, which is stored as an unbounded string.
PRINCIPAL	The name of the client principal that invoked the request, which is stored as an unbounded string.
TARGET	The object reference of the target object.

You can call PODREQ only once for each operation invocation. PODREQ must be called after a request has been dispatched to a server, and before any calls are made to access the parameter values. The DISPINIT include member contains a call to STRGET to retrieve the operation name from the reqinfo data item. You can make similar calls to retrieve the other variables in reqinfo.

Parameters

The parameter for `PODREQ` can be described as follows:

`request_details`	This is an `in` parameter that contains a PL/I structure representing the current request.

Example

The example can be broken down as follows:

1. The following code is in the *`idlmembername`*`I` server implementation module, generated by the Orbix IDL compiler (where *`idlmembername`* represents the name of the IDL member that contains the IDL definitions):

```
/* Entry point to enable the Orbix PL/I runtime to call  */
/* out to the server implementation for when a request   */
/* comes in.                                             */
DISPTCH: ENTRY;
```

2. The following code is in the supplied `DISPINIT` include member that the server implementation includes:

```
/* reqinfo is used to store information about the current */
/* request                                                */
dcl 1 reqinfo,
      3 interface_name    ptr               init(sysnull()),
      3 operation_name    ptr               init(sysnull()),
      3 principal         ptr               init(sysnull()),
      3 target            ptr               init(sysnull());

dcl operation             char(256);
dcl operation_length      fixed bin(31)     init(256);

/* Retrieve the information about the current request     */
/* received                                               */
call podreq(reqinfo);
if check_errors('podreq') ^= completion_status_yes then
   return;

/* We can now retrieve the operation name of this request  */
call strget(operation_name,
            operation,
            operation_length);
if check_errors('strget') ^= completion_status_yes then
   return;
```

3. The `select` statement in the `SELECT` include member then calls the appropriate server implementation procedure.

Exceptions

A `CORBA::BAD_INV_ORDER::NO_CURRENT_REQUEST` exception is raised if there is no request currently in progress.

A `CORBA::BAD_INV_ORDER::SERVER_NAME_NOT_SET` exception is raised if `PODSRVR` is not called.

See also

"STRGET" on page 397.

PODRUN

Synopsis

```
PODRUN
// Indicates the server is ready to accept requests.
```

Usage

Server-specific.

Description

The `PODRUN` function indicates that a server is ready to start receiving client requests. It is equivalent to calling `ORB::run()` in C++. See the *CORBA Programmer's Reference, C++* for more details about `ORB::run()`. There are no parameters required for calling `PODRUN`.

Parameters

`PODRUN` takes no parameters.

Example

In the *`idlmembername`*`V` module (that is, the server mainline member), which the Orbix IDL compiler generates, the final PL/I runtime call is a call to `PODRUN`. `PODRUN` is called after the server has written its IOR to a member, as shown in the following example:

```
/* Write out the IOR for each interface */
open file(IORFILE) output;

call objget(grid_obj,
            iorrec,iorrec_len);
if check_errors('objget') ^= completion_status_yes then return;
read file(IORFILE) into(iorrec);

close file(IORFILE);

/* Server is now ready to accept requests */
call podrun(server_name,server_name_len);
if check_errors('podrun') ^= completion_status_yes then return;
```

Exceptions

A `CORBA::BAD_INV_ORDER::SERVER_NAME_NOT_SET` exception is raised if `PODSRVR` is not called.

PODSRVR

Synopsis

```
PODSRVR(in CHAR(*) server_name,
        in FIXED BIN(31) server_name_length)
// Sets the server name for the current server process.
```

Usage

Server-specific.

Description

The PODSRVR function sets the server name for the current server. You must call this only once in a server, and it must be called before PODRUN.

Parameters

The parameters for PODSRVR can be described as follows:

server_name	This is an in parameter that is a bounded string containing the server name.
server_name_length	This is an in parameter that specifies the length of the string containing the server name.

Example

The following code is based on the generated code for the `simple` server demonstration, with extra comments for clarity:

```
dcl srv_name                          char(256) var;
dcl server_name                       char(256);
dcl server_name_len                   fixed bin(31);
…
/* Server name srv_name is read in from a file */
server_name     = srv_name;
server_name_len = length(srv_name);
…
/* Initialize the server connection to the ORB */
call orbargs(arg_list,arg_list_len,orb_name,orb_name_len);
if check_errors('orbargs') ^= completion_status_yes then return;

/* Call podsrvr using the server name passed in */
call podsrvr(server_name,server_name_len);
if check_errors('podsrvr') ^= completion_status_yes then return;

/*  Register interface  :  simple */
call podregi(addr(simple_interface),
             simple_obj);
if check_errors('podregi') ^= completion_status_yes then return;

/* Write out the IOR for the interface */
…

/* Server is now ready to accept requests */
call podrun(server_name,server_name_len);
if check_errors('podrun') ^= completion_status_yes then return;

…
```

Exceptions

A `CORBA::BAD_INV_ORDER::SERVER_NAME_ALREADY_SET` exception is raised if `ORBSRVR` is called more than once.

PODSTAT

Synopsis

```
PODSTAT(in PTR status_buffer)
// Registers the status information structure.
```

Usage

Common to clients and servers.

Description

The `PODSTAT` function registers the supplied status information structure to the PL/I runtime. The status of any PL/I runtime call is then available for examination, for example, to test if a call has completed successfully. You should call `PODSTAT` before any other PL/I runtime call. The address of the status structure is passed to `PODSTAT`. After each subsequent call to the PL/I runtime, a call to `CHECK_ERRORS` should be made to test the completion status of the call.

You should call `PODSTAT` in every program. For a client, it should be called in the main module. For a server, it should be called in the server mainline (that is, the *`idlmembername`*`V` module generated by the Orbix IDL compiler). If you do not call `PODSTAT`, no status information is available. Also, if an exception occurs and `PODSTAT` has not been called, the program terminates unless either of the following applies:

- Storage has been assigned to `POD_STATUS_INFORMATION`, which ensures that `COMPLETION_STATUS` always equals zero (that is, no error).
- No calls to `check_errors` are made.

If neither of the preceding scenarios apply when an exception occurs at runtime, and you have not called `PODSTAT`, the application terminates with the following message:

```
An exception has occurred but PODSTAT has not been called.
Place the PODSTAT API call in your application, compile and
rerun. Exiting now.
```

If you need to access the status information from other PL/I modules that might be linked into your client or server, use `PODINFO` to retrieve the stored pointer to the original `POD_STATUS_INFORMATION` data structure. You can then access the status information as usual.

Parameters

The parameter for PODSTAT can be described as follows:

status_buffer	This is an in parameter that contains a PL/I representation of the status information block structure. This buffer is populated when a CORBA system exception occurs during subsequent API calls.

Example

The Orbix IDL compiler generates the following code in the *idlmembername*V (that is, server mainline) module:

```
%include CORBA;

…

alloc pod_status_information set(pod_status_ptr);

call podstat(pod_status_ptr);
if check_errors('podstat') ^= completion_status_yes then return;
```

Exceptions

If a CORBA exception is raised, the CORBA_EXCEPTION, COMPLETION_STATUS, and EXCEPTION_MINOR_CODE field is set to non-zero. You can use the CHECK_ERRORS function to test for this. The CORBA include member lists the values that the CORBA_EXCEPTION field can be set to.

Definition

POD_STATUS_INFORMATION is defined in the CORBA include member. For example:

```
/*
    (EXTRACT FROM CORBA)
    EXCEPTION_TEXT is a pointer to the text of the exception.
    STRGET must be used to extract this text.
*/
DCL POD_STATUS_PTR                PTR;
DCL 1 POD_STATUS_INFORMATION      BASED(POD_STATUS_PTR),
      3 CORBA_EXCEPTION           FIXED BIN(15) INIT(0),
      3 COMPLETION_STATUS         FIXED BIN(15) INIT(0),
      3 EXCEPTION_MINOR_CODE      FIXED BIN(31) INIT(0),
      3 EXCEPTION_TEXT            PTR           INIT(SYSNULL());

DCL COMPLETION_STATUS_YES         FIXED BIN(15) INIT(0) STATIC;
DCL COMPLETION_STATUS_NO          FIXED BIN(15) INIT(1) STATIC;
DCL COMPLETION_STATUS_MAYBE       FIXED BIN(15) INIT(2) STATIC;
```

A CORBA::BAD_INV_ORDER::STAT_ALREADY_CALLED exception is raised if PODSTAT is called more than once.

See also

"CHECK_ERRORS" on page 426.

PODTIME

Synopsis

```
PODTIME(in FIXED BIN(15) timeout_type
        in FIXED BIN(31) timeout_value)
// Used by clients for setting the call timeout.
// Used by servers for setting the event timeout.
```

Usage

Common to batch clients and servers. Not relevant to CICS or IMS.

Description

The `PODTIME` function provides:

- Call timeout support to clients. This means that it specifies how long before a client should be timed out after having established a connection with a server. In this case, the value set by `PODSTAT` is ignored when making a connection between a client and server. The value only comes into effect after the connection has been established.
- Event timeout support to servers. This means that it specifies how long a server should wait between connection requests.

Parameters

The parameters for `PODTIME` can be described as follows:

`timeout_type`	This is an `in` parameter that specifies whether call timeout or event timeout functionality is required. It must be set to one of the two values defined in the `CORBA` include member for `POD_EVENT_TIMEOUT` or `POD_CALL_TIMEOUT`. In this case, value `1` corresponds to event timeout, and value `2` corresponds to call timeout.
`timeout_value`	This is an `in` parameter that specifies the timeout value in milliseconds.

Server example

On the server side, PODTIME must be called immediately before calling PODRUN. After PODRUN has been called, the event timeout value cannot be changed. For example:

```
…
/* Set the event timeout value to two minutes  */
call podtime(pod_event_timeout,120000);
if check_errors('podtime') ^= completion_status_yes then return;

call podrun;
if check_errors('podrun') ^= completion_status_yes then return;
```

Client example

On the client side, PODTIME must be called before calling PODEXEC. For example:

```
…
/* Set the call timeout value to thirty seconds  */
call podtime(pod_call_timeout,30000);
if check_errors('podtime') ^= completion_status_yes then return;

call podexec(…);
if check_errors('podexec') ^= completion_status_yes then return;
```

Exceptions

A CORBA::BAD_PARAM::INVALID_TIMEOUT_TYPE exception is raised if the timeout_type parameter is not set to one of the two values defined for POD_EVENT_TIMEOUT or POD_CALL_TIMEOUT in the CORBA include member.

PODVER

Overview

```
PODVER(out CHAR(*) runtime_id_version,
       out CHAR(*) runtime_compile_time_date)
// Returns PL/I runtime compile-time information.
```

Usage

Common to clients and servers.

Description

The PODVER function is used to determine which version of the PL/I runtime is being used to compile Orbix PL/I programs, because this information is not provided by the ordinary PL/I runtime libraries (that is, those without debugging output).

Parameters

The parameters for PODVER can be described as follows:

runtime_id_version	This is an out parameter that specifies the PL/I runtime ID and version. It is 14 characters in length and takes the following format: POD2000 v6.0.*x*
runtime_compile_time_date	This is an out parameter that specifies compile time and date information. It is 20 characters in length and takes the following format: *MMM DD YYYY* at *xx*:*xx*

Example

The following code example shows how a client or server can call PODVER:

```
dcl getpodver                          char(14);
dcl getpoddate                         char(20);

call podver(getpodver,getpoddate);

put skip list('pod type and version = ' || getpodver);
put skip list ('pod compile date & time = ' || getpoddate);
```

SEQALOC

Synopsis

```
SEQALOC(out PTR sequence_control_data,
        in FIXED BIN(31) sequence_size,
        in CHAR(*) typecode_key,
        in FIXED BIN(31) typecode_key_length)
// Allocates memory for an unbounded sequence.
```

Usage

Common to clients and servers.

Description

The SEQALOC function allocates initial storage for an unbounded sequence. You must call SEQALOC before you call SEQSET for the first time. The length supplied to the function is the initial sequence size requested. The typecode supplied to SEQALOC must be the sequence typecode.

You can use SEQALOC only on unbounded sequences.

Parameters

The parameters for SEQALOC can be described as follows:

sequence_control_data	This is an inout parameter that contains the unbounded sequence control data.
sequence_size	This is an in parameter that specifies the maximum expected size of the sequence.
typecode_key	This is an in parameter that contains a PL/I structure representing the typecode key. This is a bounded string.
typecode_key_length	This is an in parameter that specifies the length of the typecode key.

Note: The typecode keys are defined in the *idlmembername*T include member, and are suffixed with _tc.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    typedef sequence<long>    seqlong;
    attribute seqlong         myseq;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *idlmembername*T module (where *idlmembername* represents the name of the IDL member that contains the IDL definitions):

```
/* Extract from EXAMPLT  */
dcl 1 example_myseq_type based,
    3 result,
        5 result_seq       ptr              init(sysnull()),
        5 result_buf       fixed bin (31)  init(0);
```

   Based on the preceding IDL, the Orbix IDL compiler also generates the following code, in the *idlmembername*M module::

```
/* Extract from EXAMPLM  */
dcl 1 example_myseq_attr aligned like example_myseq_type;
```

3. The following example shows how the code generated in the *idlmembername*T and *idlmembername*M modules can then be used by the user's implementation code in the *idlmembername*I module:

```
/* Extract from EXAMPLI showing some of the user's  */
/* implementation. Allocate space for 20 elements   */
/* in the unbounded sequence myseq.                 */
call seqaloc(example_myseq_attr.result.result_seq,
             20,
             example_seqlong_tc,
             length(example_seqlong_tc));
```

Exceptions

A `CORBA::NO_MEMORY` exception is raised if there is not enough memory available to complete the request. In this case, the pointer will contain a null value.

A CORBA::BAD_PARAM::INVALID_SEQUENCE exception is raised if the sequence has not been set up correctly.

See also

- "SEQFREE" on page 375.
- "SEQSET" on page 390.
- "Memory Handling" on page 275.

SEQDUPL

Synopsis

```
SEQDUPL(in PTR sequence_control_data,
        out PTR dupl_seq_control_data)
// Duplicates an unbounded sequence control block.
```

Usage

Common to clients and servers.

Description

The SEQDUPL function creates a copy of an unbounded sequence. The new sequence has the same attributes as the original sequence. The sequence data is copied into a newly allocated buffer. The program owns this allocated buffer. When this buffer is no longer required, SEQFREE must be called to release the storage allocated to it.

You can call SEQDUPL only on unbounded sequences.

Parameters

The parameters for SEQDUPL can be described as follows:

sequence_control_data	This is an in parameter that contains the unbounded sequence control data. The address of the buffer is passed to SEQDUPL.
dupl_seq_control_data	This is an out parameter that contains the duplicated unbounded sequence control data block.

Example

The following is an example of how to use SEQDUPL in a client or server program (the example is based on two unbounded sequences of float types—that is, sequence<float> in IDL):

```
dcl 1 example_seq_args aligned,
    3 result,
        5 result_seq                 ptr,
        5 result_buf                 float dec(6);

dcl 1 example_seq_2_args aligned,
    3 result,
        5 result_seq                 ptr,
        5 result_buf                 float dec(6);

/* seqaloc step for example_seq_args and seqset is omitted  */
…
call seqdupl(example_seq_args.result.result_seq,
             example_seq_2_args.result.result_seq);
…
```

Exceptions

A CORBA::BAD_PARAM::INVALID_SEQUENCE exception is raised if the sequence has not been set up correctly.

See also

- “SEQFREE” on page 375.
- “Memory Handling” on page 275.

SEQFREE

Synopsis

```
SEQFREE(in PTR sequence_control_data)
// Frees the memory allocated to an unbounded sequence.
```

Usage

Common to clients and servers.

Description

The `SEQFREE` function releases storage assigned to a sequence. (Storage is assigned to a sequence by calling `SEQALOC` or `SEQINIT`.) Do not try to use the sequence again after freeing its memory, because doing so might result in a runtime error. Memory leaks can occur if you do not call `SEQFREE` in a logical order of innermost nested sequence to outermost.

You can use `SEQFREE` both on bounded and unbounded sequences.

Parameters

The parameter for `SEQFREE` can be described as follows:

`sequence_control_data`	This is an `in` parameter that contains the unbounded sequence control data.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    typedef sequence<long,10>    seqlong10;
    attribute seqlong10          myseq;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *`idlmembername`*`T` module (where *`idlmembername`* represents the name of the IDL member that contains the IDL definitions):

```
/* Extract from EXAMPLT  */
dcl 1 example_myseq_type based,
    3 result,
        5 result_seq       ptr             init(sysnull()),
        5 result_dat(10)   fixed bin (31)  init((10)0);
```

Based on the preceding IDL, the Orbix IDL compiler also generates the following code, in the *idlmembername*M module:

```
/* Extract from EXAMPLM  */
dcl 1 example_myseq_attr aligned like example_myseq_type;
```

3. The following example shows how the code generated in the *idlmembername*T and *idlmembername*M modules can be used by the user's implementation code in the *idlmembername*I module:

```
/* Extract from EXAMPLI showing some of the user's        */
/* implementation. Our unbouded sequence gets initialized */
/* with 25 elements' space                                */
call seqaloc(example_myseq_attr.result.result_seq,
             25,
             example_seqlong_tc,
             length(example_seqlong_tc));
if check_errors('seqaloc') ^= completion_status_yes then
   return;

/* Processing omitted */
…

/* Finished working with the unbounded sequence, now     */
/* free it                                               */
call seqfree(example_myseq.result.result_seq);
```

See also

- "SEQALOC" on page 370.
- "Memory Handling" on page 275.

SEQGET

Synopsis

```
SEQGET(in PTR sequence_control_data,
       in FIXED BIN(31) element_number,
       out PTR sequence_data)
// Retrieves the specified element from an unbounded sequence.
```

Usage

Common to clients and servers.

Description

The `SEQGET` function provides access to a specific element of an unbounded sequence. The data is copied from the specified element into the supplied sequence buffer (that is, into the `sequence_data` parameter).

You can use `SEQGET` only on unbounded sequences.

Parameters

The parameters for `SEQGET` can be described as follows:

`sequence_control_data`	This is an `in` parameter that contains the unbounded sequence control data.
`element_number`	This is an `in` parameter that specifies the index of the element number to be retrieved.
`sequence_data`	This is an `out` parameter that contains the buffer to which the sequence data is to be copied.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    typedef sequence<long>       seqlong;
    attribute seqlong            myseq;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *idlmembername*T module (where *idlmembername* represents the name of the IDL member that contains the IDL definitions):

```
/* Extract from EXAMPLT  */
dcl 1 example_myseq_type based,
    3 result,
        5 result_seq          ptr            init(sysnull()),
         5 result_buf           fixed bin (31)  init(0);
```

3. Based on the preceding IDL, the Orbix IDL compiler also generates the following code, in the *idlmembername*M module:

```
/* Extract from EXAMPLM  */
dcl 1 example_myseq_attr aligned like example_myseq_type;
```

4. The following example shows how the code generated in the *idlmembername*T and *idlmembername*M modules can then be used by the user's implementation code in a client program:

```
/* Extract from a client showing some of the user’s       */
/* implementation                                          */
dcl (i, myseq_len, myseq_value)  fixed bin(31)  init(0);

/* Retrieve the contents of attribute myseq               */
call podexec(example_obj,
             example_get_myseq,
             addr(example_myseq_args),
             no_user_exceptions);
if check_errors('podexec') ^= completion_status_yes then
   return;

/* Find out how many elements of myseq’s sequence have    */
/* been set                                                */
call seqlen(example_myseq_attr, myseq_len);
if check_errors('seqlen') ^= completion_status_yes then
   return;

put skip list('Number of results returned:', myseq_len);

/* Display the contents of each element in the attribute  */
do i = 1 to myseq_len;
    call seqget(example_myseq_args.result.result_seq, i,
                myseq_value);
    put skip list('Element', i, ' contains', myseq_value);
end;
```

Exceptions

A `CORBA::BAD_PARAM::INVALID_SEQUENCE` exception is raised if the sequence has not been set up correctly.

A `CORBA::BAD_PARAM::INVALID_BOUNDS` exception is raised if the element to be accessed is either set to `0` or greater than the current length.

See also

"SEQSET" on page 390.

SEQINIT

Synopsis

```
SEQINIT(out PTR sequence_control_data,
        in CHAR(*) typecode_key,
        in FIXED BIN(31) typecode_key_length)
// Initializes a bounded sequence
```

Usage

Common to clients and servers.

Description

The SEQINIT function initializes a bounded sequence. It sets the maximum and current length to the size of the bounded sequence, and it sets the sequence typecode to be the same as the typecode supplied to SEQINIT. The sequence data buffer is set to null. If you want to fill only part of the sequence, you can use SEQLSET to indicate how many items of the sequence have been filled.

You must supply SEQINIT with the sequence typecode.

SEQINIT can be used only on bounded sequences.

Parameters

The parameters for SEQINIT can be described as follows:

sequence_control_data	This is an out parameter that contains the unbounded sequence control data.
typecode_key	This is an in parameter that contains a PL/I structure representing the typecode key. This is a bounded string.
typecode_key_length	This is an in parameter that specifies the length of the typecode key.

Exceptions

A CORBA::BAD_PARAM::INVALID_SEQUENCE exception is raised if an unbounded sequence is passed to SEQINIT.

A CORBA::BAD_PARAM::UNKNOWN_TYPECODE exception is raised if an invalid typecode is passed to SEQINIT.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    typedef sequence<long,10>       seqlong10;
    attribute seqlong10             myseq;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *idlmembername*T module (where *idlmembername* represents the name of the IDL member that contains the IDL definitions):

```
/* Extract from EXAMPLT  */
dcl 1 example_myseq_type based,
    3 result,
       5 result_seq          ptr            init(sysnull()),
        5 result_dat(10)       fixed bin (31)  init((10)0);
```

   Based on the preceding IDL, the Orbix IDL compiler also generates the following code, in the *idlmembername*M module:

```
/* Extract from EXAMPLM  */
dcl 1 example_myseq_attr aligned like example_myseq_type;
```

3. The following example shows how the code generated in the *idlmembername*T and *idlmembername*M modules can then be used by the user's implementation code in the *idlmembername*I module:

```
/* Extract from EXAMPLI showing some of the user's        */
/* implementation. Initialize our bounded sequence before */
/* we use it.                                             */
call seqinit(example_myseq_attr.result.result_seq,
             example_seqlong10_tc,
             length(example_seqlong10_tc));
```

See also

"SEQLSET" on page 384.

SEQLEN

Synopsis

```
SEQLEN(in PTR sequence_control_data,
       out FIXED BIN(31) sequence_size)
// Retrieves the current length of the sequence
```

Usage

Common to clients and servers.

Description

The `SEQLEN` function retrieves the current length of a given bounded or unbounded sequence.

You can call `SEQLEN` both on bounded and unbounded sequences.

Parameters

The parameters for `SEQLEN` can be described as follows:

`sequence_control_data`	This is an `in` parameter that contains the unbounded sequence control data.
`sequence_size`	This is an `out` parameter that specifies the maximum size of the sequence.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    typedef sequence<long>          seqlong;
    attribute seqlong               myseq;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *`idlmembername`*`T` module (where *`idlmembername`* represents the name of the IDL member that contains the IDL definitions):

```
/* Extract from EXAMPLT  */
dcl 1 example_myseq_type based,
    3 result,
       5 result_seq          ptr            init(sysnull()),
        5 result_buf          fixed bin (31)  init(0);
```

Based on the preceding IDL, the Orbix IDL compiler also generates the following code, in the *idlmembername*M module:

```
/* Extract from EXAMPLM  */
dcl 1 example_myseq_attr aligned like example_myseq_type;
```

3. The following example shows how the code generated in the *idlmembername*T and *idlmembername*M modules can then be used by the user's implementation code in a client program:

```
/* Extract from a client showing some of the user's       */
/* implementation                                          */
dcl (i, myseq_len, myseq_value) fixed bin(31)   init(0);

/* Retrieve the contents of attribute myseq                */
call podexec(example_obj, example_get_myseq,
             addr(example_myseq_args), no_user_exceptions);
if check_errors('podexec') ^= completion_status_yes then
   return;

/* Find out how many elements of myseq's sequence have    */
/* been set                                                */
call seqlen(example_myseq_attr, myseq_len);
if check_errors('seqlen') ^= completion_status_yes then
   return;

put skip list('Number of results returned:', myseq_len;
```

Exceptions

A CORBA::BAD_PARAM::INVALID_SEQUENCE exception is raised if a null pointer is supplied to SEQLEN.

See also

"SEQMAX" on page 387.

SEQLSET

Synopsis

```
SEQLSET(in PTR sequence_control_data,
        in FIXED BIN(31) new_sequence_size)
// Changes the number of elements in the sequence
```

Usage

Common to clients and servers.

Description

The SEQLSET function resizes a sequence. The parameter for the new length of the sequence can have any value between 0 and the current length of the sequence plus one. However, it cannot be larger than the maximum length for the sequence. If a sequence is made smaller, the contents of the elements greater than the new length of the sequence are undefined.

SEQLSET is typically used when the full bounds of a bounded sequence, or the full allocation of an unbounded sequence, is not needed for storing a set of results. It can also be used for setting a sequence to a length of zero, to indicate, for example, that no records match a query.

You can call SEQLSET both on bounded and unbounded sequences.

Parameters

The parameters for SEQLSET can be described as follows:

sequence_control_data	This is an in parameter that contains the unbounded sequence control data.
new_sequence_size	This is an out parameter that specifies the new maximum size of the sequence.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    typedef sequence<long,10>       seqlong10;
    attribute seqlong10             myseq;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *idlmembername*T module (where *idlmembername* represents the name of the IDL member that contains the IDL definitions):

```
/* Extract from EXAMPLT  */
dcl 1 example_myseq_type based,
    3 result,
       5 result_seq          ptr            init(sysnull()),
        5 result_dat(10)       fixed bin (31)  init(10)0);
```

Based on the preceding IDL, the Orbix IDL compiler also generates the following code, in the *idlmembername*M module:

```
/* Extract from EXAMPLM  */
dcl 1 example_myseq_attr aligned like example_myseq_type;
```

3. The following example shows how the code generated in the *idlmembername*T and *idlmembername*M modules can be used by the user's implementation code in the *idlmembername*I module:

```
/* Extract from EXAMPLI showing some of the user's         */
/* implementation. A simple example where the user asks    */
/* for a set of powers of a given number                   */
dcl base_number                fixed bin(31);
dcl number_of_entries          fixed bin(31);

/* Initialization and misc processing omitted              */
…

base_number = 4;
number_of_entries = 6;

/* Resize the sequence to be of size number_of_entries.    */
/* This is done to facilitate the client. The client will  */
/* call SEQLEN and process just the returned number of     */
/* entries, not the entire bounded sequence, unless it     */
/* is fully filled.                                        */
call seqlset(example_myseq_args.result.result_seq,
             number_of_entries);
if check_errors('seqlset') ^= completion_status_yes then
   return;

do i = 1 to number_of_entries;
   example_myseq_attr.result.result_dat(i) = base_number**i;
end;
```

Exceptions

A `CORBA::BAD_PARAM::INVALID_SEQUENCE` exception is raised if an attempt is made to set any element to be greater than either the current length of the sequence plus one or the maximum length defined for the sequence, or if a null sequence is passed to `SEQLSET`.

See also

"SEQMAX" on page 387.

SEQMAX

Synopsis

```
SEQMAX(in PTR sequence_control_data,
       out FIXED BIN(31) max_sequence_size)
// Returns the maximum set length of the sequence
```

Usage

Common to clients and servers.

Description

The `SEQMAX` function retrieves the current maximum length of a given sequence. In the case of a bounded sequence, the current maximum length is set to the bounded size. In the case of an unbounded sequence, the current maximum length is at least the size of the initial number of elements declared for the unbounded sequence (for example, through `SEQALOC`).

You can call `SEQMAX` both on bounded and unbounded sequences.

Parameters

The parameters for `SEQMAX` can be described as follows:

`sequence_control_data`	This is an `in` parameter that contains the unbounded sequence control data.
`max_sequence_size`	This is an `out` parameter that specifies the maximum size of the sequence.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    typedef sequence<long>           seqlong;
    attribute seqlong                myseq;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *idlmembername*T module (where *idlmembername* represents the name of the IDL member that contains the IDL definitions):

```
/* Extract from EXAMPLT  */
dcl 1 example_myseq_type based,
    3 result,
       5 result_seq          ptr            init(sysnull()),
        5 result_buf           fixed bin (31)  init(0);
```

Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *idlmembername*M module:

```
/* Extract from EXAMPLM  */
dcl 1 example_myseq_attr aligned like example_myseq_type;
```

3. The following example shows how the code generated in the *idlmembername*T and *idlmembername*M modules can be used by the user's implementation in the *idlmembername*I module:

```
/* Extract from EXAMPLI showing some of the user’s         */
/* implementation                                          */
dcl myseq_length                fixed bin(31)   init(0);

/* Initialize our unbounded sequence with 25 elements     */
call seqaloc(myseq_args.result.result_seq,
             25,
             useqlong_tc,
             length(useqlong_tc));
if check_errors('seqaloc') ^= completion_status_yes then
   return;

…

/* Check what the maximum length of the sequence is now.   */
/* Note that it may not necessarily be 25 - if more than   */
/* 25 elements were set in the sequence, the maximum       */
/* length will be dynamically increased to cater for the   */
/* longer sequence.                                        */

call seqmax(example_myseq_attr.result_seq, myseq_length);
if check_errors('seqmax') ^= completion_status_yes then
   return;

put skip list ('Present maximum length of myseq =',
   myseq_length);
```

Exceptions

A `CORBA::BAD_PARAM::INVALID_SEQUENCE` exception is raised if a null pointer is supplied to `SEQMAX`.

See also

- “SEQALOC” on page 370.
- “SEQLEN” on page 382.

SEQSET

Synopsis

```
SEQSET(in PTR sequence_control_data,
       in FIXED BIN(31) element_number,
       in PTR sequence_data)
// Places the specified data into the specified element of an
// unbounded sequence.
```

Usage

Common to clients and servers.

Description

The `SEQSET` function copies the supplied data into the requested element of an unbounded sequence. You can set any element ranging between `1` and the current length of a sequence plus one. If the current length plus one is greater than the maximum size of the sequence, the sequence is reallocated to hold the enlarged sequence.

You can call `SEQSET` only on unbounded sequences.

The algorithm used by SEQSET to determine the new maximum size of the sequence, whenever necessary, is:

```
max_seq_size = SEQMAX(sequence_control_data)

if element_number > max_seq_size then
    if max_seq_size < 8192 then
        new_max_seq_size = max_seq_size * 2
    else
        new_max_seq_size = max_seq_size + (max_seq_size/8)
    end
end
```

Parameters

The parameters for `SEQSET` can be described as follows:

`sequence_control_data`	This is an `in` parameter that contains the unbounded sequence control data.
`element_number`	This is an `in` parameter that specifies the index of the element number that is to be set.
`sequence_data`	This is an `in` parameter that contains the buffer containing the data that is to be placed in the sequence.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    typedef sequence<long>           seqlong;
    attribute seqlong                myseq;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *idlmembername*T module (where *idlmembername* represents the name of the IDL member that contains the IDL definitions):

```
/* Extract from EXAMPLT  */
dcl 1 example_myseq_type based,
    3 result,
       5 result_seq          ptr            init(sysnull()),
        5 result_buf           fixed bin (31)  init(0);
```

3. Based on the preceding IDL, the Orbix IDL compiler also generates the following code, in the *idlmembername*M module:

```
/* Extract from EXAMPLM  */
dcl 1 example_myseq_attr aligned like example_myseq_type;
```

4. The following example shows how the code generated in the *idlmembername*T and *idlmembername*M modules can be used by the user's implementation code in the *idlmembername*I module:

```
/* Extract from EXAMPLI showing some of the user's          */
/* implementation                                            */
dcl element_num                fixed bin(31);
dcl max_seq_ele                fixed bin(31);

/* Set up the sequence to hold 10 elements                   */
max_seq_ele = 10;
call seqaloc(example_myseq_args.result.result_seq,
             max_seq_ele);

/* Set each element of the unbounded sequence with           */
/* multiples of 12                                           */
do element_num = 1 to max_seq_ele;
    example_myseq_args.result.result_buf = element_num*12;

    call seqset(example_myseq_arts.result.result_seq,
                element_num,
                addr(example_myseq_args.result.result_buf));
    if check_errors('seqset') ^= completion_status_yes then
        return;
end;
```

Exceptions

A CORBA::BAD_PARAM::INVALID_SEQUENCE exception is raised if the sequence has not been set up correctly. For example, if an invalid sequence typecode was passed to SEQSET or if the sequence is a bounded sequence.

A CORBA::BAD_PARAM::INVALID_BOUNDS exception is raised if the element to be accessed is either set to 0 or greater than the current maximum length of the sequence plus one.

A CORBA::NO_MEMORY exception is raised if the sequence needs to be resized and there is not enough memory to resize it.

See also

"SEQGET" on page 377.

STRCON

Synopsis

```
STRCON(inout PTR string_pointer,
       in PTR addon_string_pointer)
// Concatenates two unbounded strings.
```

Usage

Common to clients and servers.

Description

The `STRCON` function concatenates the two supplied unbounded strings, and returns the concatenated unbounded string in the first parameter. The original storage allocated to the first string pointer is deleted, because it is assigned the concatenated string instead.

Parameters

The parameters for `STRCON` can be described as follows:

`string_pointer`	This is an `inout` parameter that is the unbounded string pointer containing a copy of the bounded string that is to be modified. This string is subsequently returned with the `addon_string_pointer` string appended to it.
`addon_string_pointer`	This is an in parameter that contains the string to be concatenated to the other string supplied in `string_pointer`.

Example

1. Consider the following test program:

```
TEST: PROC OPTIONS(MAIN);

dcl first_part                          ptr;
dcl second_part                         ptr;
dcl temp_string                         char(40) init('');
dcl temp_string_len                     fixed bin(31) init(40);

temp_string = 'Hello ';
call strset(first_part, temp_string, temp_string_len);

temp_string = 'There';
call strset(second_part, temp_string, temp_string_len);

call strcon(first_part, second_part);

temp_string = '';
call strget(first_part, temp_string, temp_string_len);

put skip list('Contents of first_part are: ', temp_string);

END TEST;
```

2. The results that are printed from this test program are as follows:

```
Contents of first_part are: Hello There
```

STRDUPL

Synopsis

```
STRDUPL(in PTR string_pointer,
        out PTR duplicate_string_pointer)
// Duplicates a given unbounded string
```

Usage

Common to clients and servers.

Description

The STRDUPL function takes in an unbounded string as its first parameter, duplicates the string, and returns it via its second parameter. This involves a complete copy (that is, the storage used by the in string is also duplicated).

Parameters

The parameters for STRDUPL can be described as follows:

string_pointer	This is an in parameter that is the unbounded string pointer containing a copy of the bounded string.
duplicate_string_pointer	This is an out parameter that contains the duplicated string.

Example

The following is an example of how to use STRDUPL in a client or sever program:

```
dcl orig_str_ptr              PTR;
dcl dupl_str_ptr              PTR;
dcl temp_string               char(40) init('hello');
dcl temp_string_len           fixed bin (31) init(40);

/* Set up our first string     */
call strset(orig_str_ptr, temp_string, temp_string_len);
if check_errors('strset') ^= completion_status_yes then return;

/* Make a copy of orig_str_ptr, storing this in           */
/* dupl_str_ptr                                           */
call strdupl(orig_str_ptr, dupl_str_ptr);
if check_errors('strdupl') ^= completion_status_yes then return;
```

STRFREE

Synopsis

```
STRFREE(in PTR string_pointer)
// Frees the storage used by an unbounded string
```

Usage

Common to clients and servers.

Description

The STRFREE function releases dynamically allocated memory for an unbounded string, via a pointer that was originally obtained by calling STRSET. Do not try to use the unbounded string after freeing it, because doing so might result in a runtime error.

Parameters

The parameter for STRFREE can be described as follows:

string_pointer	This is an in parameter that is the unbounded string pointer containing a copy of the unbounded string.

Example

The following is an example of how to use STRFREE in a client or server program:

```
dcl unb_string          ptr        init(sysnull());
dcl pli_string          char(32)   init('Orbix');

call strset(unb_string, pli_string, length(pli_string));
if check_errors('strset') ^= completion_status_yes then return;
…

/* Retrieve the string from the unbounded string  */
pli_string='';
call strget(unb_string, pli_string, length(pli_string));
put skip list('The string set was: ' || pli_string);

/* Finished using the unbounded string now, so free it  */
call strfree(unb_string);
```

See also

- "STRSET" on page 401.
- "Memory Handling" on page 275.

STRGET

Synopsis

```
STRGET(in PTR string_pointer,
       out CHAR(*) string,
       in FIXED BIN(31) string_length)
// Copies the contents of an unbounded string to a PL/I string
```

Usage

Common to clients and servers.

Description

The `STRGET` function copies the characters from the unbounded string pointer to the PL/I string item. If the unbounded string does not contain enough characters to fill the PL/I string exactly, the PL/I string is padded with spaces. If the length of the unbounded string is greater than the size of the PL/I string, only the length specified by the third parameter is copied into the PL/I string from the string pointer.

> **Note:** Null characters are never copied from the `string_pointer` to the target string.

Parameters

The parameters for `STRGET` can be described as follows:

`string_pointer`	This is an `in` parameter that is the unbounded string pointer containing a copy of the unbounded string.
`string`	This is an `out` parameter that is a bounded string to which the contents of the string pointer are copied. This string is terminated by a space if it is larger than the contents of the string pointer.
`string_length`	This is an `in` parameter that specifies the length of the unbounded string.

Example

1. Consider the following test program:

```
TEST: PROC OPTIONS(MAIN);

%include CORBA;

/* Temporary string used to set a string in src_pointer  */
dcl temp_string                   char(32) init('Hello there');

/* This is the supplied PL/I unbounded string pointer  */
dcl str_pointer             ptr;

/* This is the PL/I representation of the string       */
dcl dest                    char(64);

/* Set up the src_pointer unbounded string             */
call strset(str_pointer, temp_string, length(temp_string));
if check_errors('strset') ^= completion_status_yes then return;

/* Our call to strget will now retrieve the string stored  */
/* in str_pointer and set the dest PL/I string             */
call strget(str_pointer, dest, length(dest));
if check_errors('strget') ^= completion_status_yes then return;

put skip list('Contents of str_pointer: ' || dest);

END TEST;
```

2. The results printed out from the preceding test program are:

```
Contents of str_pointer: Hello there
```

See also

“STRSET” on page 401.

STRLENG

Synopsis

```
STRLENG(in PTR string_pointer,
        out FIXED BIN(31) string_length)
// Returns the actual length of an unbounded string
```

Usage

Common to clients and servers.

Description

The `STRLENG` function returns the number of characters in an unbounded string.

Parameters

The parameters for `STRLENG` can be described as follows:

`string_pointer`	This is an `in` parameter that is the unbounded string pointer containing the unbounded string.
`string_length`	This is an `out` parameter that is used to retrieve the actual length of the string that the `string_pointer` contains.

Example

1. Consider the following test program:

```
TEST: PROC OPTIONS(MAIN);

%include CORBA;

dcl str_ptr                ptr;
dcl len                    fixed bin(31);
dcl temp_string            char(32);

temp_string = 'This is a string';
call strset(str_ptr, temp_string, length(temp_string));
if check_errors('strset') ^= completion_status_yes then return;

/* Call strleng and store the result in len       */
call strleng(str_ptr, len);
if check_errors('strleng') ^= completion_status_yes then return;

put skip list('The length of our unbounded string is', len);

END TEST;
```

2. The results printed out from the preceding test program are:

```
The length of our unbounded string is     16
```

STRSET

Synopsis

```
STRSET(out PTR string_pointer,
       in CHAR(*) string,
       in FIXED BIN(31) string_length)
// Creates an unbounded string from a CHAR(n) data item.
```

Usage

Common to clients and servers.

Description

The STRSET function creates an unbounded string, and copies the number of characters specified in the third parameter for the PL/I string's length from the PL/I string to the unbounded string. If the PL/I string contains trailing spaces, these are not copied to the unbounded string.

Note: STRSET allocates memory for the string from the program heap at runtime. See "STRFREE" on page 396 and "Unbounded Strings and Memory Management" on page 285 for details of how this memory is subsequently released.

Parameters

The parameters for STRSET can be described as follows:

`string_pointer` This is an `out` parameter to which the unbounded string is copied.

`string` This is an `in` parameter containing the bounded string that is to be copied. This string is terminated by a space if it is larger than the contents of the target string pointer. If the bounded string contains trailing spaces, they are not copied.

`string_length` This is an `in` parameter that specifies the number of characters to be copied from the bounded string specified in `string`.

Example

1. Consider the following test program:

```
TEST: PROC OPTIONS(MAIN);

% include CORBA;

dcl string_one_ptr            PTR;
dcl string_two_ptr            PTR;
dcl temp_string               CHAR(64);
dcl len                       FIXED BIN(31);

temp_string = 'This is a string     ';

/* Set the first unbounded string with STRSET     */
call strset(string_one_ptr, temp_string, length(temp_string));
if check_errors('strset') ^= completion_status_yes then return;

/* Set the second unbounded string with STRSETS    */
call strsets(string_two_ptr, temp_string, length(temp_string));
if check_errors('strset') ^= completion_status_yes then return;

/* Retrieve the length of both strings             */
call strleng(string_one_ptr, len);
if check_errors('strleng') ^= completion_status_yes then return;
put skip list('The length of String 1 is', len);

call strleng(string_two_ptr, len);
if check_errors('strleng') ^= completion_status_yes then return;
put skip list('The length of String 2 is', len);

END TESTSTR;
```

2. The following results are displayed after running the preceding test program:

```
THE LENGTH OF STRING 1 IS                16
THE LENGTH OF STRING 2 IS                20
```

See also

- "STRFREE" on page 396.
- "STRGET" on page 397 .
- "Unbounded Strings and Memory Management" on page 285.

STRSETS

Synopsis

```
STRSETS(out PTR string_pointer,
        in CHAR(*) string,
        in FIXED BIN(31) string_length)
// Creates an unbounded string from a CHAR(n) data item
```

Usage

Common to clients and servers.

Description

The STRSETS function is exactly the same as STRSET, except that STRSETS does copy trailing spaces to the unbounded string. See "STRSET" on page 401 for more details.

Note: STSETS allocates memory for the string from the program heap at runtime. See "STRFREE" on page 396 and "Unbounded Strings and Memory Management" on page 285 for details of how this memory is subsequently released.

See also

- "STRGET" on page 397.
- "Unbounded Strings and Memory Management" on page 285.

STR2OBJ

Synopsis

```
STR2OBJ(in PTR object_string,
        out PTR object_reference)
// Creates an object reference from an interoperable object
// reference (IOR).
```

Usage

Common to clients and servers.

Description

The `STR2OBJ` function creates an object reference from an unbounded string. When a client has called `STR2OBJ` to create an object reference, the client can then invoke operations on the server.

Parameters

The parameters for `STR2OBJ` can be described as follows:

`object_string`	This is an `in` parameter that contains a pointer to the address in memory where the interoperable object reference is held. This parameter can take different forms. See "Format for input string" for more details.
`object_reference`	This is an `out` parameter that contains a pointer to the address in memory where the returned object reference is held.

Format for input string

The `object_string` input parameter can take different forms, as follows:

- Stringified interoperable object reference (IOR)

  The CORBA specification defines the representation of stringified IOR references, so this form is interoperable across all ORBs that support IIOP. For example:

  ```
  IOR:000…
  ```

  You can use the supplied `iordump` utility to parse the IOR. The `iordump` utility is available with your Orbix Mainframe installation on OS/390 UNIX System Services.

- corbaloc:rir URL

This is one of two possible formats relating to the corbaloc mechanism. The corbaloc mechanism uses a human-readable string to identify a target object. A corbaloc:rir URL can be used to represent an object reference. It defines a key upon which `resolve_initial_references` is called (that is, it is equivalent to calling `OBJRIR`).

The format of a corbaloc:rir URL is `corbaloc:rir:/`*`rir-argument`* (for example, `"corbaloc:rir:/NameService"`). See the *CORBA Programmer's Guide, C++* for more details on the operation of `resolve_initial_references`.

- corbaloc:iiop-address URL

  This is the second of two possible formats relating to the corbaloc mechanism. A corbaloc:iiop-address URL is used to identify named-keys.

  The format of a corbaloc:iiop-address URL is `corbaloc:`*`iiop-address`*`[,`*`iiop-address`*`]`…`/`*`key-string`* (for example, `"corbaloc:iiop:xyz.com/BankService"`).

- itmfaloc URL

  The itmfaloc URL facilitates locating IMS and CICS adapter objects. Using an itmfaloc URL is similar to using the `itadmin mfa resolve` command; except that the itmfaloc URL exposes this functionality directly to Orbix applications.

  The format of an itmfaloc URL is `itmfaloc:`*`itmfaloc-argument`* (for example, `"itmfaloc:Simple/SimpleObject"`). See the *CICS Adapters Administrator's Guide* and the *IMS Adapters Administrator's Guide* for details on the operation of itmfaloc URLs.

Stringified IOR example

Consider the following example of a client program that first shows how the server's object reference is retrieved via STR2OBJ, and then shows how the object reference is subsequently used:

```
dcl IORFILE                            file stream;
dcl iorrec                             char(2048)    init(' ');
dcl iorrec_len                         fixed bin(31) init(2048);
dcl iorrec_ptr                         ptr           init(sysnull());

…

/* Read in the IOR from a file  */
get file(IORFILE) edit(iorrec) (column (1), a(iorrec_len));
close file(IORFILE);

/* Create an unbounded IOR string */
call strset(iorrec_ptr, iorrec, iorrec_len);
if check_errors('strset') ^= completion_status_yes then return;

/* Create an object reference now using the unbounded IOR */
/* string */
call str2obj(iorrec_ptr, Simple_SimpleObject_obj);
if check_errors('objset') ^= completion_status_yes then return;

/* We are now ready to invoke operations on the server  */
call podexec(Simple_SimpleObject_obj,
             Simple_SimpleObject_call_me,
             addr(Simple_SimpleObject_c_ba77_args),
             no_user_exceptions);
if check_errors('podexec') ^=completion_status_yes then return;
```

corbaloc:rir URL example

Consider the following example that uses a corbaloc to call `resolve_initial_references` on the Naming Service:

```
dcl corbaloc_str char(26) init ('corbaloc:rir:/NameService ');
dcl corbaloc_ptr ptr init(sysnull());
dcl naming_service_obj ptr init(sysnull());

/* Create an unbounded corbaloc string to Naming Service */
call strset(corbaloc_ptr, corbaloc_str, length(corbaloc_str));
if check_errors('strset') ^= completion_status_yes then return;

/* Create an object reference using the unbounded corbaloc str */
call str2obj(corbaloc_ptr, naming_service_obj);

/* Can now invoke on naming service */
```

corbaloc:iiop-address URL example

You can use `STR2OBJ` to resolve a named key. A named key, in essence, associates a string identifier with an object reference. This allows access to the named key via the string identifier. Named key pairings are stored by the locator. The following is an example of how to create a named key:

```
itadmin named_key create -key TestObjectNK IOR:…
```

Consider the following example that shows how to use `STR2OBJ` to resolve this named key:

```
dcl corbaloc_str char(46)
    init ('corbaloc:iiop:1.2@localhost:5001/TestObjectNK ');
dcl corbaloc_ptr ptr init(sysnull());
dcl test_object_obj ptr init(sysnull());

/* Create an unbounded corbaloc string to the Test Object */
call strset(corbaloc_ptr, corbaloc_str, length(corbaloc_str));
if check_errors('strset') ^= completion_status_yes then return;

/* Create an object reference using the unbounded corbaloc str */
call str2obj(corbaloc_ptr, test_object_obj);

/* Can now invoke on TestObject */
```

itmfaloc URL example

You can use STR2OBJ to locate IMS and CICS server objects via the itmfaloc mechanism. To use an itmfaloc URL, ensure that the configuration scope used contains a valid initial reference for the adapter that is to be used. You can do this in either of the following ways:

- Ensure that the LOCAL_MFA_REFERENCE in your Orbix configuration contains an object reference for the adapter you want to use.
- Use either "-ORBname iona_services.imsa" or "-ORBname iona_services.cicsa" to explicitly pass across a domain that defines IT_MFA initial references.

Consider the following example that shows how to locate IMS and CICS server objects via the itmfaloc URL mechanism:

```
dcl corbaloc_str char(29)
    init ('itmfaloc:Simple/SimpleObject ');
dcl corbaloc_ptr ptr init(sysnull());
dcl test_object_obj ptr init(sysnull());

/* Create an unbounded corbaloc string to the              */
/* Simple/SimpleObject interface defined to an IMS/CICS    */
/* adapter                                                 */
call strset(corbaloc_ptr, corbaloc_str, length(corbaloc_str));
if check_errors('strset') ^= completion_status_yes then return;

/* Create an object reference using the unbounded corbaloc str */
call str2obj(corbaloc_ptr, test_object_obj);

/* Can now invoke on Simple/SimpleObject */
```

See also

"OBJ2STR" on page 335.

TYPEGET

Synopsis

```
TYPEGET(in PTR any_pointer,
        out CHAR(*) typecode_key,
        in FIXED BIN(31) typecode_key_length)
// Extracts the type name from an any.
```

Usage

Common to clients and servers.

Description

The `TYPEGET` function returns the typecode of the value of the `any`. You can then use the typecode to ensure that the correct buffer is passed to the `ANYGET` function for extracting the value of the `any`.

Parameters

The parameters for `TYPEGET` can be described as follows:

`any_pointer`	This is an `inout` parameter that is a pointer to the address in memory where the `any` is stored.
`typecode_key`	This is an `out` parameter that contains a PL/I structure to which the typecode key is copied. This is a bounded string.
`typecode_key_length`	This is an `in` parameter that specifies the length of the typecode key.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    attribute any myany;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *idlmembername*T module (where *idlmembername* represents the name of the IDL member that contains the IDL definitions):

```
/* Extract from EXAMPLT  */
dcl 1 example_myany_attr aligned,
    3 result                  ptr;
```

Based on the preceding IDL, the Orbix IDL compiler also generates the following code, in the *idlmembername*M module:

```
/* Extract from EXAMPLM  */
dcl 1 example_myany_attr aligned like example_myany_type;
```

3. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *idlmembername*I module:

```
/* Extract from EXAMPLI showing some of the user's        */
/* implementation                                          */
dcl short_value                fixed bin(15);
dcl long_value                 fixed bin(31);

call typeget(example_myany_attr.result,
             example_typecode,
             example_typecode_length);
if check_errors('typeget') ^= completion_status_yes then
   return;
select(example_typecode);
    when(corba_type_short)
        do;
           call anyget(example_myany_attr.result,
                       addr(short_value));
          if check_errors('anyget') ^= completion_status_yes
               then return;

           put skip list ('Short from ANY is', short_value);
        end;
    when(corba_type_long)
        do;
           call anyget(example_myany_attr.result,
                       addr(long_value));
           if check_errors('anyget') ^=completion_status_yes
               then return;

           put skip list('Long from ANY is', long_value);
        end;
     otherwise
        put skip list ('No SELECT case defined to extract the
            ANY');
end;
```

Exceptions

A CORBA::BAD_INV_ORDER::TYPESET_NOT_CALLED exception is raised if the typecode of the any has not been set via TYPESET.

See also

- "ANYGET" on page 317.
- "ANYSET" on page 319.

TYPESET

Synopsis

```
TYPESET(in PTR any_pointer,
        in CHAR(*) typecode_key,
        in FIXED BIN(31) typecode_key_length)
// Sets the type name of an any
```

Usage

Common to clients and servers.

Description

The TYPESET function sets the type of the any to the supplied typecode. You must call TYPESET before you call ANYSET, because ANYSET uses the current typecode information to insert the data into the any.

Parameters

The parameters for TYPESET can be described as follows:

any_pointer	This is an in parameter that is a pointer to the address in memory where the any is stored.
typecode_key	This is an in parameter containing the typecode string representation.
typecode_key_length	This is an in parameter that specifies the length of the typecode string.

Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {
    attribute any myany;
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the *idlmembername*T module (where *idlmembername* represents the name of the IDL member that contains the IDL definitions):

```
/* Extract from EXAMPLT  */
dcl 1 example_myany_type aligned,
    3 result                  ptr;
```

Based on the preceding IDL, the Orbix IDL compiler also generates the following code, in the *idlmembername*M module:

```
/* Extract from EXAMPLM  */
dcl 1 example_myany_attr aligned like example_myany_type;
```

3. The following example shows how the code generated in the *idlmembername*T and *idlmembername*M modules can be used by the user's implementation code in the *idlmembername*I module.

```
/* Extract from EXAMPLI showing some of the user's        */
/* implementation                                         */
dcl short_value                fixed bin(15);

/* Set up our value and typecode for the ANY  */
short value = 12;
example_type_code = corba_type_short;

/* Now we are ready to set the ANY myany  */
call typeset(example_myany_attr.result,
             example_typecode,
             example_typecode_length);

call anyset(example_myany_attr.result, addr(short_value));
if check_errors('anyset') ^= completion_status_yes then
   return;
```

Exceptions

A CORBA::BAD_PARAM::UNKNOWN_TYPECODE exception is raised if the typecode cannot be determined from the typecode key passed to TYPESET.

See also

- "ANYGET" on page 317.
- "ANYSET" on page 319.
- "TYPEGET" on page 409.

WSTRCON

Synopsis

```
WSTRCON(inout PTR widestring_pointer,
        in PTR addon_widestring_pointer)
// Concatenates two unbounded wide strings.
```

Usage

Common to clients and servers.

Description

The `WSTRCON` function concatenates the two supplied unbounded wide strings, and returns the concatenated unbounded wide string for the first parameter. The original storage allocated to the first wide string pointer is deleted, because it is assigned the concatenated wide string instead.

Parameters

The parameters for `WSTRCON` can be described as follows:

`widestring_pointer`	This is an `inout` parameter that is the unbounded string pointer containing a copy of the bounded wide string. This wide string is subsequently returned with the `addon_widestring_pointer` wide string appended to it.
`addon_widestring_pointer`	This is an `in` parameter that contains the wide string to be concatenated to the other wide string supplied in `string_pointer`.

Example

1. Consider the following test program:

```
TEST: PROC OPTIONS(MAIN);

dcl first_part              ptr;
dcl second_part             ptr;
dcl temp_graphic            graphic(40) init('');
dcl temp_graphic_len        fixed bin(31) init(40);
dcl temp_string             char(40) init('');

temp_graphic = graphic('Hello ');
call wstrset(first_part, temp_graphic, temp_graphic_len);

temp_graphic = graphic('There');
call wstrset(second_part, temp_graphic, temp_graphic_len);
call wstrcon(first_part, second_part);

temp_graphic = graphic('');
call wstrget(first_part, temp_graphic, temp_graphic_len);

temp_string = char(temp_graphic);
put skip list('Contents of first_part are: ', temp_string);

END TEST;
```

2. The results printed by the preceding test program are as follows:

```
Contents of first_part are: Hello There
```

WSTRDUP

Synopsis

```
WSTRDUP(in PTR widestring_pointer,
        out PTR duplicate_widestring_pointer)
// Duplicates a given unbounded wide string.
```

Usage

Common to clients and servers.

Description

The `WSTRDUP` function takes an unbounded wide string as a first parameter, duplicates the string, and then returns the duplicate wide string via its second parameter. This involves a complete copy (that is, the storage used by the `in` wide string is also duplicated).

Parameters

The parameters for `WSTRDUP` can be described as follows:

`widestring_pointer`	This is an `in` parameter that is the unbounded string pointer containing a copy of the unbounded wide string.
`duplicate_widestring_pointer`	This is an `out` parameter that contains the duplicated wide string.

Example

Consider the following example:

```
dcl orig_widestr_ptr      ptr;
dcl dupl_widestr_ptr      ptr;
dcl temp_graphic          graphic(40) init(graphic('hello'));
dcl temp_graphic_len      fixed bin(31) init(40);

/* Set up our first wide string  */
call wstrset(orig_widestr_ptr, temp_graphic,
             temp_graphic_len);
if check_errors('wstrset') ^= completion_status_yes then return;

/* Make a copy of orig_widestr_ptr,  */
/* storing it in dupl_widestr_ptr    */
call wstrdup(orig_widestr_ptr, dupl_widestr_ptr);
if check_errors('wstrdup') ^= completion_status_yes then return;
```

WSTRFRE

Synopsis

```
WSTFRE(in PTR widestring_pointer)
// Frees the storage used by an unbounded wide string.
```

Usage

Common to clients and servers.

Description

The `WSTRFRE` function releases dynamically allocated memory for an unbounded wide string, via a pointer that was originally obtained by calling `WSTRSET`. Do not try to use the unbounded wide string after freeing it, because doing so might result in a runtime error.

Parameters

The parameter for `WSTRFRE` can be described as follows:

`widestring_pointer`	This is an `in` parameter that is the unbounded wide string pointer containing a copy of the unbounded wide string.

Example

The following is an example of how to use WSTRFRE in a client or server program:

```
TSTWSTR:  PROC OPTIONS(MAIN);

%include CORBA;

dcl wstring_ptr                  PTR;
dcl temp_graphic                 GRAPHIC(64);

temp_graphic = graphic('This is a graphic ');
call wstrset(wstring_ptr, temp_graphic);
…

/* Retrieve the string from the unbounded wide string */
call wstrget(wstring_ptr, temp_graphic,
             length(temp_graphic));
put skip list('The string set was: ' || char(temp_graphic));

/* Finished using the unbounded wide string, so free it */
call wstrfre(wstring_ptr);

END TSTWSTR;
```

See also

"WSTRSET" on page 423.

WSTRGET

Synopsis

```
WSTRGET(in PTR widestring_pointer,
        out GRAPHIC(*) widestring,
        in FIXED BIN(31) widestring_length)
// Copies the contents of an unbounded wide string to a PL/I
// graphic.
```

Usage

Common to clients and servers.

Description

The `WSTRGET` function copies the characters in the incoming unbounded wide string pointer to the PL/I graphic string item. If the unbounded wide string does not contain enough characters to fill the `GRAPHIC` wide string exactly, the `GRAPHIC` wide string is padded with spaces. If the length of the wide string is greater than the size of the `GRAPHIC` wide string, only the length specified by the third parameter is copied into the `GRAPHIC` wide string from the unbounded wide string pointer. The third parameter specifies the maximum number of graphic characters that the `GRAPHIC` wide string can hold.

Null characters are never copied from the wide string to the `GRAPHIC` wide string.

Parameters

The parameters for `WSTRGET` can be described as follows:

`widestring_pointer`	This is an `in` parameter that is the unbounded string pointer containing a copy of the unbounded wide string.
`widestring`	This is an `out` parameter that is a bounded wide string to which the contents of `string_pointer` are copied. This string is terminated by a space if it is larger than the contents of `string_pointer`.
`widestring_length`	This is an `in` parameter that specifies the length of the unbounded wide string.

Example

The example can be broken down as follows:

1. Consider the following test program:

```
TEST: PROC OPTIONS(MAIN);

%include CORBA;

/* Temporary graphic used to set the wide string is  */
/* wide_str_pointer                                   */
dcl temp_graphic   graphic(32) init(graphic('Hello there'));

/* Temporary string used for retrieving data from    */
/* the graphic                                        */
dcl temp_string    char(32)    init('');

/* This is the supplied PL/I unbounded wide string   */
/* pointer                                            */
dcl wide_str_pointer   ptr;

Set up the wide_str_pointer unbounded string     */
call wstrset(wide_str_pointer, temp_graphic,
             length(temp_graphic));
if check_errors('wstrset') ^= completion_status_yes then
   return;

/* Our call to wstrget will now retrieve the graphic  */
/* stored in wide_str_pointer and set temp_graphic    */
temp_graphic = '';
call wstrget(wide_str_pointer, temp_graphic,
             length(temp_graphic));
if check_errors('wstrget') ^= completion_status_yes then
   return;

temp_string = character(temp_graphic);
put skip list('Contents of wide_str_pointer: ' ||
   temp_string);

END TEST;
```

2. The preceding test program prints the following results:

```
Contents of wide_str_pointer: Hello There
```

See also

"WSTRSET" on page 423.

WSTRLEN

Synopsis

```
WSTRLEN(in PTR widestring_pointer,
        out FIXED BIN(31) widestring_length)
/ Returns the number of characters held in the wide string
// (excluding trailing nulls).
```

Usage

Common to clients and servers.

Description

The `WSTRLEN` function returns the number of characters in an unbounded wide string.

Parameters

The parameters for `WSTRLEN` can be described as follows:

`widestring_pointer`	This is an `in` parameter that is the unbounded wide string pointer containing the unbounded wide string.
`widestring_length`	This is an `out` parameter that is used to retrieve the actual length of the wide string that the `widestring_pointer` contains.

Example

1. Consider the following test program:

```
TEST: PROC OPTIONS(MAIN);

%include CORBA;

dcl wide_str_ptr              ptr;
dcl len                       fixed bin(31);
dcl temp_graphic              graphic(32);

temp_graphic = graphic('This is a graphic');

call wstrset(wide_str_ptr, temp_graphic,
             length(temp_graphic));
if check_errors('wstrset') ^= completion_status_yes then
   return;

/* Call wstrlen and store the result in len   */
call wstrlen(wide_str_ptr, len);
if check_errors('wstrlen') ^= completion_status_yes then
   return;

put skip list('The length of our unbounded wide string is',
   len);

END TEST;
```

2. The preceding program prints the following results:

```
The length of our unbounded wide string is   17
```

WSTRSET

Synopsis

```
WSTRSET(out PTR widestring_pointer,
        in GRAPHIC(*) widestring,
        in FIXED BIN(31) widestring_length)
// Creates an unbounded wide string from a GRAPHIC(n) data item
```

Usage

Common to clients and servers.

Description

The WSTRSET function creates an unbounded wide string, and then copies the number of graphic characters specified in the third parameter from the GRAPHIC wide string to the wide string. WSTRSET does not copy trailing spaces to the wide string, if they are present in the GRAPHIC wide string.

Parameters

The parameters for WSTRSET can be described as follows:

widestring_pointer	This is an out parameter to which the unbounded wide string is copied.
widestring	This is an in parameter containing the bounded wide string that is to be copied. This string is terminated by a space if it is larger than the contents of the target string pojnter. If the bounded wide string contains trailing spaces, they are not copied.
widestring_length	This is an in parameter that specifies the number of characters to be copied from the bounded wide string specified in string.

Example

1. Consider the following test program:

```
TSTWSTR: PROC OPTIONS(MAIN);

%include CORBA;

dcl wstring_one_ptr              ptr;
dcl wstring_two_ptr              ptr;
dcl temp_graphic                 graphic(64);
dcl len                          fixed bin(31);

temp_graphic = graphic('This is a graphic      ');

/* Set the first unbounded wide string with WSTRSET   */
call wstrset(wstring_one_ptr, temp_graphic,
             length(temp_graphic));
if check_errors('wstrset') ^= completion_status_yes then
   return;

/* Set the second unbounded wide string with WSTRSTS   */
call strsets(wstring_two_ptr, temp_graphic,
             length(temp_graphic));
if check_errors('wstrsts') ^= completion_status_yes then
   return;

/* Retrieve the length of both wide strings            */
call wstrlen(wstring_one_ptr, len);
if check_errors('wstrlen') ^= completion_status_yes then
   return;
put skip list('The length of wide string 1 is', len);

call wstrlen(wstring_two_ptr, len);
if check_errors('wstrlen') ^= completion_status_yes then
   return;
put skip list('The length of wide string 2 is', len);

END TSTWSTR;
```

2. The preceding test program displays the following results:

```
The length of wide string 1 is            17
The length of wide string 2 is            20
```

See also

"WSTRGET" on page 419.

WSTRSTS

Synopsis

```
WSTRSTS(out PTR widestring_pointer,
        in GRAPHIC(*) widestring,
        in FIXED BIN(31) widestring_length)
// Creates an unbounded wide string from a GRAPHIC(n) data item
```

Usage

Common to clients and servers.

Description

The `WSTRSTS` function is exactly the same as `WSTRSET`, except that `WSTRSTS` does copy trailing spaces to the unbounded wide string. See "WSTRSET" on page 423 for more details.

See also

"WSTRGET" on page 419.

CHECK_ERRORS

Synopsis

```
CHECK_ERRORS(in CHAR(*) function_name)
    RETURNS(FIXED BIN(31) error_number)
// Tests the completion status of the last PL/I runtime call.
```

Usage

Common to clients and servers.

Description

The CHECK_ERRORS helper function tests whether the most recent call to the PL/I runtime completed successfully. It is not part of the PL/I runtime itself. CHECK_ERRORS examines the corba_exception variable in the pod_status_information structure, which is updated after every PL/I runtime call. If a CORBA exception has not occurred, CHECK_ERRORS returns a completion_status value of zero; if a CORBA exception has occurred, CHECK_ERRORS returns a completion_status value of other than zero. If the completion_status value is not zero, a message is displayed showing details about the error that occurred.

The completion_status value is stored in the pod_status_information structure in the CORBA include member. This return value can be used to determine whether or not to continue processing. The completion_status value can be one of the following:

- COMPLETION_STATUS_YES corresponds to value 0.
- COMPLETION_STATUS_NO corresponds to value 1.
- COMPLETION_STATUS_MAYBE corresponds to value 2.

Parameters

The parameters for CHECK_ERRORS can be described as follows:

function_name	This is an in parameter that contains the name of the function being called.
error_number	This is a return parameter that contains the error number pertaining to the error raised.

Definition

The `CHECK_ERRORS` function is defined as follows in the `CHKERRS` include member:

```
/* Determine the system exception name from the exception     */
/* number                                                     */
 %include EXCNAME;

/*************************************************************/
/* Function : CHECK_ERRORS                                    */
/* Purpose  : Test the last PL/I Runtime call for system      */
/* exceptions                                                 */
/*************************************************************/
 CHECK_ERRORS: PROC(FUNCTION_NAME) RETURNS(FIXED BIN(15));

 dcl function_name                  char(*);
 dcl sysprint                       ext file stream print output;
 dcl exception_len                  fixed bin(31);
 dcl exception_info                 char(*) ctl;
 dcl pliretc                        builtin;

 call podinfo(pod_status_ptr);

 if pod_status_information.corba_exception ^= 0 then
   do;
     call strleng(pod_status_information.exception_text,
          exception_len);
     alloc exception_info char(exception_len);
     call strget(pod_status_information.exception_text,
          exception_info,exception_len);

     put skip list('System Exception encountered');
     put skip list('Function called  :',function_name);
     put skip list('Exception name   :',
        corba_exc_name(pod_status_information.corba_exception));
     put skip list('Exception        :',exception_info);
     free exception_info;

     call pliretc(12);  /* set the return code for the program */
   end;

 return(pod_status_information.completion_status);

 END CHECK_ERRORS;
```

Note: The CHKERRS include member is used in server and batch client programs. It is replaced with CHKCLCIC in CICS client programs, and CHKCLIMS in IMS client programs. See Table 6 on page 52 and Table 14 on page 96 for more details of these include members.

Example

The following is an example of how to use CHECK_ERRORS in a program:

```
call strset(orig_str_ptr, input_string, length(input_string));
if check_errors('strset') ^= completion_status_yes then return;
```

Deprecated and Removed APIs

Deprecated APIs

This section summarizes the APIs that were available with the Orbix 2.3 PL/I adapter, but which are now deprecated with the Orbix PL/I runtime.

```
OBJGET(PTR,                  /* IN : object reference         */
       CHAR(*),              /* OUT: IOR reference            */
       FIXED BIN(31));       /* IN : IOR reference length     */
// Orbix 2.3 : Returned an interoperable object reference (IOR).
// Orbix Mainframe: Replaced with OMG PL/I function OBJ2STR.
//                  Works as in Orbix 2.3.x.

OBJGETM(PTR,                  /* IN : object reference         */
        CHAR(*),              /* OUT: marker name              */
        FIXED BIN(31));       /* IN : marker name length       */
// Orbix 2.3 : Retrieves the marker name from an object reference.
// Orbix Mainframe: Replaced with OMG PL/I function OBJGTID.
//                  Retrieves the object ID from an IOR.

OBJSET(CHAR(*),              /* IN : object name              */
       PTR);                 /* OUT: object reference         */
// Creates an object reference from a stringified Orbix object
// reference.

OBJSETM(CHAR(*),              /* IN : object name              */
        CHAR(*),              /* IN : marker                   */
        PTR);                 /* OUT: object reference         */
// Creates an object reference from a stringified Orbix object
// reference and sets its marker.

PODALOC(PTR,                  /* OUT: pointer to memory block   */
        FIXED BIN(31));       /* IN : amount of memory required */
// Orbix 2.3 : Allocated memory.
// Orbix Mainframe: Replaced with OMG PL/I function MEMALOC.
//                  Performs the same function as in Orbix 2.3.

PODEBUG(PTR,                  /* IN : pointer to memory         */
        FIXED BIN(15),        /* IN : size of memory dump       */
        CHAR(*),              /* IN : explanatory text string   */
        FIXED BIN(15));       /* IN : length of text string     */
// Orbix 2.3 : Output a formatted memory dump for the specified
//             block of memory.
// Orbix Mainframe: Replaced with OMG PL/I function MEMDBUG.
//                  Performs the same function as in Orbix 2.3.
```

```
PODFREE(PTR);                  /* IN : pointer to memory block   */
// Orbix 2.3 : Freed the specified block of memory.
// Orbix Mainframe: Replaced with OMG PL/I function MEMDBUG.
//                  Performs the same function as in Orbix 2.3.

PODHOST(CHAR(*),               /* OUT: hostname length           */
        FIXED BIN(31));        /* IN : hostname                  */
// Orbix 2.3 : Returned hostname of server.
// Orbix Mainframe: Not required by Orbix PL/I servers.

PODINIT(CHAR(*),               /* IN : server name               */
        FIXED BIN(31));        /* IN : server name length        */
// Orbix 2.3 : Equivalent to calling ORB::run() in C++. Parameters
//             supplied to PODINIT are ignored.
// Orbix Mainframe: Replaced with PODRUN.

PODRASS(FIXED BIN(31),         /* IN : minor error number        */
        FIXED BIN(15));        /* IN : completion status         */
// Orbix 2.3 : Signalled a user exception to Orbix via return
//             codes.
// Orbix Mainframe: Replaced with PODERR. Throws a system
//                  exception.

PODREGI(PTR,                   /* IN : interface description     */
        PTR);                  /* OUT: object reference          */
// Orbix 2.3 : Described an interface to the PL/I runtime,
//             returning an IOR.
// Orbix Mainframe: Superceded by using PODREG and OBJNEW.
```

Removed APIs

This section summarizes the APIs that are no longer available with Orbix PL/I.

```
OBJGETO(PTR,               /* IN : object reference              */
        CHAR(*),           /* OUT: Orbix object reference        */
        FIXED BIN(31));    /* IN : Orbix object reference length */
// Orbix 2.3 : Returns a stringified Orbix object reference.
// Orbix Mainframe: Not supported because Orbix protocol not
//                  supported.

OBJLEN(PTR,                /* IN : IOR string                    */
       FIXED BIN(31));     /* OUT: length of object reference    */
OBJLENO(PTR,               /* IN : object reference              */
        FIXED BIN(31));    /* OUT: length of object reference    */
// Orbix 2.3 : Returns the length of an object reference.
// Orbix Mainframe: Not supported. Not required for Orbix
//                  Mainframe.
```

```
PODEXEC(PTR,               /* IN   : object reference           */
       CHAR(*),            /* IN   : operation name             */
       PTR);               /* INOUT: address(operation_buffer)  */
// Orbix 2.3 : Invokes an operation on the object.
// Orbix Mainframe: Replaced with a new version with a fourth
//                  parameter for a user exception data field.
```

Part 3

Appendices

In this part

This part contains the following appendices:

POA Policies

This appendix summarizes the POA policies that are supported by the Orbix PL/I runtime, and the argument used with each policy.

In this appendix

This appendix contains the following sections:

Overview

A POA's policies play an important role in determining how the POA implements and manages objects and processes client requests. There is only one POA created by the Orbix PL/I runtime, and that POA uses only the policies listed in this chapter.

See the *CORBA Programmer's Guide, C++* for more details about POAs and POA policies. See the `PortableServer::POA` interface in the *CORBA Programmer's Reference*, *C++* for more details about the POA interface and its policies.

Note: The POA policies described in this chapter are the only POA policies that the Orbix PL/I runtime supports. Orbix PL/I programmers have no control over these POA policies. They are outlined here simply for the purposes of illustration and the sake of completeness.

POA policy listing

Table 46 describes the policies that are supported by the Orbix PL/I runtime, and the argument used with each policy.

Table 46: *POA Policies Supported by PL/I Runtime (Sheet 1 of 3)*

Policy	Argument Used	Description
Id Assignment	USER_ID	This policy determines whether object IDs are generated by the POA or the application. The USER_ID argument specifies that only the application can assign object IDs to objects in this POA. The application must ensure that all user-assigned IDs are unique across all instances of the same POA. USER_ID is usually assigned to a POA that has an object lifespan policy of PERSISTENT (that is, it generates object references whose validity can span multiple instances of a POA or server process, so the application requires explicit control over object IDs).
Id Uniqueness	MULTIPLE_ID	This policy determines whether a servant can be associated with multiple objects in this POA. The MULTIPLE_ID specifies that any servant in the POA can be associated with multiple object IDs.
Implicit Activation	NO_IMPLICIT_ACTIVATION	This policy determines the POA's activation policy. The NO_IMPLICIT_ACTIVATION argument specifies that the POA only supports explicit activation of servants.

Table 46: *POA Policies Supported by PL/I Runtime (Sheet 2 of 3)*

Policy	Argument Used	Description
Lifespan	PERSISTENT	This policy determines whether object references outlive the process in which they were created. The PERSISTENT argument specifies that the IOR contains the address of the location domain's implementation repository, which maps all servers and their POAs to their current locations. Given a request for a persistent object, the Orbix daemon uses the object's virtual address first, and looks up the actual location of the server process via the implementation repository.
Request Processing	USE_ACTIVE_OBJECT_MAP_ONLY	This policy determines how the POA finds servants to implement requests. The USE_ACTIVE_OBJECT_MAP_ONLY argument assumes that all object IDs are mapped to a servant in the active object map. The active object map maintains an object-servant mapping until the object is explicitly deactivated via deactivate_object(). This policy is typically used for a POA that processes requests for a small number of objects. If the object ID is not found in the active object map, an OBJECT_NOT_EXIST exception is raised to the client. This policy requires that the POA has a servant retention policy of RETAIN.

Table 46: *POA Policies Supported by PL/I Runtime (Sheet 3 of 3)*

Policy	Argument Used	Description
Servant Retention	`RETAIN`	The `RETAIN` argument with this policy specifies that the POA retains active servants in its active object map.
Thread	`SINGLE_THREAD_MODEL`	The `SINGLE_THREAD_MODEL` argument with this policy specifies that requests for a single-threaded POA are processed sequentially. In a multi-threaded environment, all calls by a single-threaded POA to implementation code (that is, servants and servant managers) are made in a manner that is safe for code that does not account for multi-threading.

System Exceptions

This appendix summarizes the Orbix system exceptions that are specific to the Orbix PL/I runtime.

> **Note:** This appendix does not describe other Orbix system exceptions that are not specific to the PL/I runtime. See the *CORBA Programmer's Guide, C++* for details of these other system exceptions.

In this appendix

This appendix contains the following sections:

CORBA::INITIALIZE:: exceptions

The following exception is defined within the `CORBA::INITIALIZE::` scope:

`UNKNOWN`	This exception is raised by any API when the exact problem cannot be determined.

CORBA::BAD_PARAM:: exceptions

The following exceptions are defined within the `CORBA::BAD_PARAM::` scope:

`UNKNOWN_OPERATION`	This exception is raised by `PODEXEC`, if the operation is not valid for the interface.
`NO_OBJECT_IDENTIFIER`	This exception is raised by `OBJNEW`, if the parameter for the object name is an invalid string.
`INVALID_SERVER_NAME`	This exception is raised if the server name that is passed does not match the server name passed to `PODSRVR`.

CORBA::INTERNAL:: exceptions

The following exceptions are defined within the `CORBA::INTERNAL::` scope:

`UNEXPECTED_INVOCATION`	This exception is raised on the server side when a request is being processed, if a previous request has not completed successfully.
`UNKNOWN_TYPECODE`	This exception is raised internally by the PL/I runtime, to show that a serious error has occurred. It normally means that there is an issue with the typecodes in relation to either the *`idlmembername`*`X` include member or the application itself.
`INVALID_STREAMABLE`	This exception is raised internally by the PL/I runtime, to show that a serious error has occurred. It normally means that there is an issue with the typecodes in relation to either the *`idlmembername`*`X` include member of the application itself.

CORBA::BAD_INV_ORDER:: exceptions

The following exceptions are defined within the `CORBA::BAD_INV_ORDER::` scope:

`INTERFACE_NOT_REGISTERED`	This exception is raised if the specified interface has not been registered via `PODREG`.
`INTERFACE_ALREADY_REGISTERED`	This exception is raised by `PODREG`, if the client or server attempts to register the same interface more than once.

ADAPTER_ALREADY_INITIALIZED	This exception is raised by ORBARGS, if it is called more than once in a client or server.
STAT_ALREADY_CALLED	This exception is raised by PODSTAT if it is called more than once.
SERVER_NAME_ALREADY_SET	This exception is raised by PODSRVR, if the API is called more than once.
SERVER_NAME_NOT_SET	This exception is raised by OBJNEW, PODREQ, OBJGTID, or PODRUN, if PODSRVR is called.
NO_CURRENT_REQUEST	This exception is raised by PODREQ, if no request is currently in progress.
ARGS_NOT_READ	This exception is raised by PODPUT, if the in or inout parameters for the request have not been processed.
ARGS_ALREADY_READ	This exception is raised by PODGET, if the in or inout parameters for the request have already been processed.
TYPESET_NOT_CALLED	This exception is raised by ANYSET or TYPEGET, if the typecode for the any type has not been set via a call to TYPESET.

Installed Data Sets

This appendix provides an overview listing of the data sets installed with Orbix Mainframe that are relevant to development and deployment of PL/I applications.

In this appendix

This appendix contains the following sections:

Overview

The list of data sets provided in this appendix is specific to PL/I and intentionally omits any data sets specific to COBOL or C++. For a full list of all installed data sets see the Mainframe Installation Guide.

List of PL/I-related data sets

Table 47 lists the installed data sets that are relevant to PL/I.

Table 47: *List of Installed Data Sets Relevant to PL/I (Sheet 1 of 4)*

Data Set	Description
orbixhlq.ADMIN.GRAMMAR	Contains `itadmin` grammar files.
orbixhlq.ADMIN.HELP	Contains `itadmin` help files.
orbixhlq.ADMIN.LOAD	Contains Orbix administration programs.
orbixhlq.CONFIG	Contains Orbix configuration information.

Table 47: *List of Installed Data Sets Relevant to PL/I (Sheet 2 of 4)*

Data Set	Description
orbixhlq.DEMOS.CICS.MFAMAP	Used to store CICS server adapter mapping member information for demonstrations.
orbixhlq.DEMOS.CICS.PLI.BLD.JCL	Contains jobs to build the CICS PL/I demonstrations.
orbixhlq.DEMOS.CICS.PLI.LOAD	Used to store programs for the CICS PL/I demonstrations.
orbixhlq.DEMOS.CICS.PLI.PLINCL	Used to store generated files for the CICS PL/I demonstrations.
orbixhlq.DEMOS.CICS.PLI.README	Contains documentation for the CICS PL/I demonstrations.
orbixhlq.DEMOS.CICS.PLI.SRC	Contains program source for the CICS PL/I demonstrations.
orbixhlq.DEMOS.IDL	Contains IDL for demonstrations.
orbixhlq.DEMOS.IMS.MFAMAP	Used to store IMS server adapter mapping member information for demonstrations.
orbixhlq.DEMOS.IMS.PLI.BLD.JCL	Contains jobs to build the IMS PL/I demonstrations.
orbixhlq.DEMOS.IMS.PLI.LOAD	Used to store programs for the IMS PL/I demonstrations.
orbixhlq.DEMOS.IMS.PLI.PLINCL	Used to store generated files for the IMS PL/I demonstrations.
orbixhlq.DEMOS.IMS.PLI.README	Contains documentation for the IMS PL/I demonstrations.
orbixhlq.DEMOS.IMS.PLI.SRC	Contains program source for the IMS PL/I demonstrations.
orbixhlq.DEMOS.IORS	Used to store IORs for demonstrations.
orbixhlq.DEMOS.PLI.BLD.JCL	Contains jobs to build the PL/I demonstrations.

Table 47: *List of Installed Data Sets Relevant to PL/I (Sheet 3 of 4)*

Data Set	Description
orbixhlq.DEMOS.PLI.LOAD	Used to store programs for the PL/I demonstrations.
orbixhlq.DEMOS.PLI.MAP	Used to store name substitution maps for the PL/I demonstrations.
orbixhlq.DEMOS.PLI.PLINCL	Used to store generated files for the PL/I demonstrations.
orbixhlq.DEMOS.PLI.README	Contains documentation for the PL/I demonstrations.
orbixhlq.DEMOS.PLI.RUN.JCL	Contains jobs to run the PL/I demonstrations.
orbixhlq.DEMOS.PLI.SRC	Contains program source for the PL/I demonstrations.
orbixhlq.DEMOS.TYPEINFO	Optional type information store.
orbixhlq.DOMAINS	Contains Orbix configuration information.
orbixhlq.INCLUDE.IT@CICS.IDL	Contains IDL files.
orbixhlq.INCLUDE.IT@IMS.IDL	Contains IDL files.
orbixhlq.INCLUDE.IT@MFA.IDL	Contains IDL files.
orbixhlq.INCLUDE.OMG.IDL	Contains IDL files.
orbixhlq.INCLUDE.ORBIX.IDL	Contains IDL files.
orbixhlq.INCLUDE.ORBIX@XT.IDL	Contains IDL files.
orbixhlq.INCLUDE.PLINCL	Contains include files for PL/I demonstrations.
orbixhlq.JCL	Contains jobs to run Orbix.
orbixhlq.LKED	Contains side-decks for the DLLs.
orbixhlq.LPA	Contains LPA eligible programs.
orbixhlq.MFA.LOAD	Contains DLLS required for deployment of Orbix programs in IMS.

Table 47: *List of Installed Data Sets Relevant to PL/I (Sheet 4 of 4)*

Data Set	Description
orbixhlq.PLI.LIB	Contains programs for Orbix PL/I support.
orbixhlq.PLICICS.LIB	Contains programs for CICS-to-CICS PL/I support.
orbixhlq.PROCS	Contains JCL procedures.
orbixhlq.RUN	Contains binaries & DLLs.

Index

M

N

O

P

R

S