Contents

List of Figures xi
List of Tables xiii
Preface xv

Part 1  Programmer’s Guide

Chapter 1  Introduction to Orbix 3
  Why CORBA? 4
    CORBA Objects 5
    Object Request Broker 7
  CORBA Application Basics 8
  Orbix Plug-In Design 9
  Orbix Application Deployment 11
    Location Domains 12
    Configuration Domains 13

Chapter 2  Getting Started in Batch 15
  Overview and Setup Requirements 16
  Developing the Application Interfaces 22
    Defining IDL Interfaces 23
    Generating PL/I Source and Include Members 24
  Developing the Server 29
    Writing the Server Implementation 30
    Writing the Server Mainline 33
    Building the Server 36
  Developing the Client 37
    Writing the Client 38
    Building the Client 41
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developing the CICS Server</td>
<td>143</td>
</tr>
<tr>
<td>Writing the Server Implementation</td>
<td>144</td>
</tr>
<tr>
<td>Writing the Server Mainline</td>
<td>147</td>
</tr>
<tr>
<td>Building the Server</td>
<td>150</td>
</tr>
<tr>
<td>Preparing the Server to Run in CICS</td>
<td>151</td>
</tr>
<tr>
<td>Developing the CICS Client</td>
<td>155</td>
</tr>
<tr>
<td>Writing the Client</td>
<td>156</td>
</tr>
<tr>
<td>Building the Client</td>
<td>161</td>
</tr>
<tr>
<td>Preparing the Client to Run in CICS</td>
<td>162</td>
</tr>
<tr>
<td>Developing the CICS Two-Phase Commit Client</td>
<td>166</td>
</tr>
<tr>
<td>Writing the Client</td>
<td>167</td>
</tr>
<tr>
<td>Building the Client</td>
<td>181</td>
</tr>
<tr>
<td>Building the Servers</td>
<td>182</td>
</tr>
<tr>
<td>Preparing the Client to Run in CICS</td>
<td>183</td>
</tr>
<tr>
<td>Running the Demonstrations</td>
<td>186</td>
</tr>
<tr>
<td>Running a Batch Client against a CICS Server</td>
<td>187</td>
</tr>
<tr>
<td>Running a CICS Client against a Batch Server</td>
<td>188</td>
</tr>
<tr>
<td>Running a CICS Two-Phase Commit Client against Batch Servers</td>
<td>189</td>
</tr>
</tbody>
</table>

**Chapter 5   IDL Interfaces**  

<table>
<thead>
<tr>
<th>IDL</th>
<th>194</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modules and Name Scoping</td>
<td>195</td>
</tr>
<tr>
<td>Interfaces</td>
<td>196</td>
</tr>
<tr>
<td>Interface Contents</td>
<td>198</td>
</tr>
<tr>
<td>Operations</td>
<td>199</td>
</tr>
<tr>
<td>Attributes</td>
<td>201</td>
</tr>
<tr>
<td>Exceptions</td>
<td>202</td>
</tr>
<tr>
<td>Empty Interfaces</td>
<td>203</td>
</tr>
<tr>
<td>Inheritance of Interfaces</td>
<td>204</td>
</tr>
<tr>
<td>Multiple Inheritance</td>
<td>205</td>
</tr>
<tr>
<td>Inheritance of the Object Interface</td>
<td>207</td>
</tr>
<tr>
<td>Inheritance Redefinition</td>
<td>208</td>
</tr>
<tr>
<td>Forward Declaration of IDL Interfaces</td>
<td>209</td>
</tr>
<tr>
<td>Local Interfaces</td>
<td>210</td>
</tr>
<tr>
<td>Valuetypes</td>
<td>211</td>
</tr>
<tr>
<td>Abstract Interfaces</td>
<td>212</td>
</tr>
</tbody>
</table>

**IDL Data Types**  

<table>
<thead>
<tr>
<th>Built-in Data Types</th>
<th>214</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extended Built-in Data Types</td>
<td>217</td>
</tr>
</tbody>
</table>
Contents

Orbix IDL Compiler Arguments 305
  Summary of the arguments 306
  Specifying Compiler Arguments 308
  -D Argument 311
  -E Argument 312
  -L Argument 314
  -M Argument 316
  -O Argument 323
  -S Argument 325
  -T Argument 326
  -V Argument 329
  -W Argument 330

Orbix IDL Compiler Configuration 331
  PL/I Configuration Variables 332
  Adapter Mapping Member Configuration Variables 338
  Providing Arguments to the IDL Compiler 341

Chapter 8 Memory Handling 345
  Operation Parameters 346
    Bounded Sequences and Memory Management 347
    Unbounded Sequences and Memory Management 351
    Unbounded Strings and Memory Management 356
    Object References and Memory Management 360
    The any Type and Memory Management 364
    User Exceptions and Memory Management 369

Memory Management Routines 371

Part 2 Programmer’s Reference

Chapter 9 API Reference 377
  API Reference Summary 378
  API Reference Details 384
    ANYFREE 387
    ANYGET 389
    ANYSET 391
    MEMALOC 393
    MEMDEBUG 394
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMFREE</td>
<td>396</td>
</tr>
<tr>
<td>OBJDUPL</td>
<td>397</td>
</tr>
<tr>
<td>OBJGTID</td>
<td>399</td>
</tr>
<tr>
<td>OBJNEW</td>
<td>401</td>
</tr>
<tr>
<td>OBJREL</td>
<td>403</td>
</tr>
<tr>
<td>OBJRIR</td>
<td>405</td>
</tr>
<tr>
<td>OBJ2STR</td>
<td>407</td>
</tr>
<tr>
<td>ORBARGS</td>
<td>409</td>
</tr>
<tr>
<td>PODERR</td>
<td>413</td>
</tr>
<tr>
<td>PODEXEC</td>
<td>418</td>
</tr>
<tr>
<td>PODGET</td>
<td>421</td>
</tr>
<tr>
<td>PODINFO</td>
<td>424</td>
</tr>
<tr>
<td>PODPUT</td>
<td>426</td>
</tr>
<tr>
<td>PODREG</td>
<td>429</td>
</tr>
<tr>
<td>PODREQ</td>
<td>431</td>
</tr>
<tr>
<td>PODRUN</td>
<td>434</td>
</tr>
<tr>
<td>PODSRVR</td>
<td>435</td>
</tr>
<tr>
<td>PODSTAT</td>
<td>437</td>
</tr>
<tr>
<td>PODTIME</td>
<td>440</td>
</tr>
<tr>
<td>PODTXNB</td>
<td>442</td>
</tr>
<tr>
<td>PODTXNE</td>
<td>443</td>
</tr>
<tr>
<td>PODVER</td>
<td>444</td>
</tr>
<tr>
<td>SEQALOC</td>
<td>445</td>
</tr>
<tr>
<td>SEQDUPL</td>
<td>448</td>
</tr>
<tr>
<td>SEQFREE</td>
<td>450</td>
</tr>
<tr>
<td>SEQGET</td>
<td>452</td>
</tr>
<tr>
<td>SEQINIT</td>
<td>455</td>
</tr>
<tr>
<td>SEQLEN</td>
<td>457</td>
</tr>
<tr>
<td>SQLSET</td>
<td>459</td>
</tr>
<tr>
<td>SEQMAX</td>
<td>462</td>
</tr>
<tr>
<td>SEQREL</td>
<td>465</td>
</tr>
<tr>
<td>SEQSET</td>
<td>467</td>
</tr>
<tr>
<td>STRCON</td>
<td>470</td>
</tr>
<tr>
<td>STRDUPL</td>
<td>472</td>
</tr>
<tr>
<td>STRFREE</td>
<td>473</td>
</tr>
<tr>
<td>STRGET</td>
<td>474</td>
</tr>
<tr>
<td>STRLENG</td>
<td>476</td>
</tr>
<tr>
<td>STRSET</td>
<td>478</td>
</tr>
<tr>
<td>STRSETS</td>
<td>480</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1: The Nature of Abstract CORBA Objects ........................................... 5
Figure 2: The Object Request Broker ......................................................... 7
Figure 3: Address Space Layout for an Orbix PL/I Application ................. 47
Figure 4: Overview of IMS Transaction Layout ...................................... 94
Figure 5: Overview of CICS Transaction Layout .................................. 167
Figure 6: Inheritance Hierarchy for PremiumAccount Interface ............ 206
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1: Supplied Code and JCL</td>
<td>17</td>
</tr>
<tr>
<td>Table 2: Supplied Include Members</td>
<td>18</td>
</tr>
<tr>
<td>Table 3: Generated Server Source Code Members</td>
<td>24</td>
</tr>
<tr>
<td>Table 4: Generated PL/I Include Members</td>
<td>25</td>
</tr>
<tr>
<td>Table 5: Supplied Code and JCL</td>
<td>52</td>
</tr>
<tr>
<td>Table 6: Supplied Include Members</td>
<td>56</td>
</tr>
<tr>
<td>Table 7: Generated PL/I Include Members</td>
<td>66</td>
</tr>
<tr>
<td>Table 8: Generated Server Source Code Members</td>
<td>68</td>
</tr>
<tr>
<td>Table 9: Generated IMS Server Adapter Mapping Member</td>
<td>69</td>
</tr>
<tr>
<td>Table 10: Generated Type Information Member</td>
<td>69</td>
</tr>
<tr>
<td>Table 11: The SIMPLEI Demonstration Module</td>
<td>72</td>
</tr>
<tr>
<td>Table 12: The SIMPLEV Demonstration Module</td>
<td>75</td>
</tr>
<tr>
<td>Table 13: The SIMPLEC Demonstration Module</td>
<td>84</td>
</tr>
<tr>
<td>Table 14: Supplied Code and JCL</td>
<td>124</td>
</tr>
<tr>
<td>Table 15: Supplied Include Members</td>
<td>128</td>
</tr>
<tr>
<td>Table 16: Generated PL/I Include Members</td>
<td>138</td>
</tr>
<tr>
<td>Table 17: Generated Server Source Code Members</td>
<td>140</td>
</tr>
<tr>
<td>Table 18: Generated CICS Server Adapter Mapping Member</td>
<td>141</td>
</tr>
<tr>
<td>Table 19: Generated CICS Server Adapter Mapping Member</td>
<td>141</td>
</tr>
<tr>
<td>Table 20: Built-in IDL Data Types, Sizes, and Values</td>
<td>214</td>
</tr>
<tr>
<td>Table 21: Extended built-in IDL Data Types, Sizes, and Values</td>
<td>217</td>
</tr>
<tr>
<td>Table 22: Mapping for Basic IDL Types</td>
<td>241</td>
</tr>
<tr>
<td>Table 23: Generated Source Code and Include Members</td>
<td>302</td>
</tr>
<tr>
<td>Table 24: CORBA Type Support Provided by -E Option</td>
<td>312</td>
</tr>
<tr>
<td>Table 25: Example of Default Generated Data Names</td>
<td>316</td>
</tr>
<tr>
<td>Table 26: Example of Level-0-Scoped Generated Data Names</td>
<td>319</td>
</tr>
</tbody>
</table>
Table 27: Example of Level-1-Scoped Generated Data Names 319
Table 28: Example of Level-2-Scoped Generated Data Names 320
Table 29: Example of Modified Mapping Names 321
Table 30: Summary of PL/I Configuration Variables 333
Table 31: Adapter Mapping Member Configuration Variables 339
Table 32: Memory Handling for IN Bounded Sequences 347
Table 33: Memory Handling for INOUT Bounded Sequences 348
Table 34: Memory Handling for OUT and Return Bounded Sequences 349
Table 35: Memory Handling for IN Unbounded Sequences 351
Table 36: Memory Handling for INOUT Unbounded Sequences 352
Table 37: Memory Handling for OUT and Return Unbounded Sequences 354
Table 38: Memory Handling for IN Unbounded Strings 356
Table 39: Memory Handling for INOUT Unbounded Strings 357
Table 40: Memory Handling for OUT and Return Unbounded Strings 358
Table 41: Memory Handling for IN Object References 360
Table 42: Memory Handling for INOUT Object References 361
Table 43: Memory Handling for OUT and Return Object References 362
Table 44: Memory Handling for IN Any Types 364
Table 45: Memory Handling for INOUT Any Types 365
Table 46: Memory Handling for OUT and Return Any Types 367
Table 47: Memory Handling for User Exceptions 369
Table 48: Summary of Common Services and Their PL/I Identifiers 405
Table 49: POA Policies Supported by PL/I Runtime 512
Table 50: List of Installed Data Sets Relevant to PL/I 519
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 2.6
- GIOP 1.2 (default), 1.1, and 1.0

Orbix Mainframe is IONA’s implementation of the CORBA standard for the z/OS 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 z/OS environment.

Supported compilers

The supported compilers are:

- IBM PL/I for MVS & VM V1R1M1.
- IBM Enterprise PL/I for z/OS V3R2.
- IBM Enterprise PL/I for z/OS V3R3.
- IBM Enterprise PL/I for z/OS V3R4.
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 z/OS 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 z/OS 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 z/OS.
- The Mainframe Migration Guide, which provides details of migration issues for users who have migrated from IONA's Orbix 2.3-based solution for z/OS 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 (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:

```c
#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:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to Orbix</td>
<td>3</td>
</tr>
<tr>
<td>Getting Started in Batch</td>
<td>15</td>
</tr>
<tr>
<td>Getting Started in IMS</td>
<td>49</td>
</tr>
<tr>
<td>Getting Started in CICS</td>
<td>121</td>
</tr>
<tr>
<td>IDL Interfaces</td>
<td>193</td>
</tr>
<tr>
<td>IDL-to-PL/I Mapping</td>
<td>235</td>
</tr>
<tr>
<td>Orbix IDL Compiler</td>
<td>293</td>
</tr>
<tr>
<td>Memory Handling</td>
<td>345</td>
</tr>
</tbody>
</table>
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 z/OS environment.

In this chapter

This chapter discusses the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why CORBA?</td>
<td>4</td>
</tr>
<tr>
<td>CORBA Application Basics</td>
<td>8</td>
</tr>
<tr>
<td>Orbix Plug-In Design</td>
<td>9</td>
</tr>
<tr>
<td>Orbix Application Deployment</td>
<td>11</td>
</tr>
</tbody>
</table>
CHAPTER 1 | Introduction to Orbix

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 z/OS 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 z/OS, 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 page 12
- Configuration Domains page 13
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

<table>
<thead>
<tr>
<th>Overview</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plug-in design</td>
<td>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.</td>
</tr>
</tbody>
</table>
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.

This chapter discusses the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview and Setup Requirements</td>
<td>16</td>
</tr>
<tr>
<td>Developing the Application Interfaces</td>
<td>22</td>
</tr>
<tr>
<td>Developing the Server</td>
<td>29</td>
</tr>
<tr>
<td>Developing the Client</td>
<td>37</td>
</tr>
<tr>
<td>Running the Application</td>
<td>42</td>
</tr>
<tr>
<td>Application Address Space Layout</td>
<td>47</td>
</tr>
</tbody>
</table>

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 z/OS. 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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“Developing the Application Interfaces” on page 22.</td>
</tr>
<tr>
<td>2</td>
<td>“Developing the Server” on page 29.</td>
</tr>
<tr>
<td>3</td>
<td>“Developing the Client” on page 37.</td>
</tr>
</tbody>
</table>

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.
Overview and Setup Requirements

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 29 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 37 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)**

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orbixhlq.DEMOS.IDL(SIMPLE)</td>
<td>This is the supplied IDL.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.PLI.SRC(SIMPLEV)</td>
<td>This is the source code for the batch server mainline module.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.PLI.SRC(SIMPLEI)</td>
<td>This is the source code for the batch server implementation module.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.PLI.SRC(SIMPLEC)</td>
<td>This is the source code for the client module.</td>
</tr>
<tr>
<td>orbixhlq.JCLLIB(LOCATOR)</td>
<td>This JCL runs the Orbix locator daemon.</td>
</tr>
<tr>
<td>orbixhlq.JCLLIB(NODEDAEM)</td>
<td>This JCL runs the Orbix node daemon.</td>
</tr>
</tbody>
</table>
Table 1: Supplied Code and JCL (Sheet 2 of 2)

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

**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 121 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)

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>orbixhlq.INCLUDE.PLINCL(CHKERRS)</code></td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table 2: Supplied Include Members (Sheet 2 of 3)

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orbixhlq.INCLUDE.PLINCL(CORBA)</td>
<td>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 &amp;client_only=’yes’; or includes the SETUPCL member.</td>
</tr>
<tr>
<td>orbixhlq.INCLUDE.PLINCL(CORBACOM)</td>
<td>This contains common PL/I runtime function definitions that can be used both by clients and servers.</td>
</tr>
<tr>
<td>orbixhlq.INCLUDE.PLINCL(CORBASV)</td>
<td>This contains PL/I runtime function definitions that can be used by servers.</td>
</tr>
<tr>
<td>orbixhlq.INCLUDE.PLINCL(DISPINIT)</td>
<td>This is used by servers. It retrieves the current request information into the REQINFO structure via POQREQ. From REQINFO the operation to be performed by the server is retrieved via a call to STRGET.</td>
</tr>
<tr>
<td>orbixhlq.INCLUDE.PLINCL(EXCNAME)</td>
<td>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.</td>
</tr>
<tr>
<td>orbixhlq.INCLUDE.PLINCL(IORREC)</td>
<td>This is used by both clients and servers. It contains declarations for storing an IOR file and its size.</td>
</tr>
<tr>
<td>orbixhlq.INCLUDE.PLINCL(READIOR)</td>
<td>This declares the file IORFILE, reads an IOR into 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 file that has been read. Additionally, it also sets up several ON_ERROR blocks that check the status of IORFILE and catch any general errors that might occur in the client.</td>
</tr>
</tbody>
</table>
### Table 2: Supplied Include Members (Sheet 3 of 3)

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orbixhlq.INCLUDE.PLINCL(SETUPCL)</td>
<td>This is relevant to clients only. It contains preprocessor statements used to prevent CORBASV from being included in client-side programs. It should be included before CORBA in every client module.</td>
</tr>
<tr>
<td>orbixhlq.INCLUDE.PLINCL(SETUPSV)</td>
<td>This is relevant to servers only. It contains preprocessor statements used to ensure CORBASV is included in server-side programs and to prevent warnings that client_only has not been declared. It should be included before CORBA in every server source module.</td>
</tr>
<tr>
<td>orbixhlq.INCLUDE.PLINCL(URLSTR)</td>
<td>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 481 for more details. <strong>Note:</strong> Even though batch applications can use this include member, the supplied batch demonstration does not use this.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.PLI.PLINCL</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Note:** Any supplied include members that are not listed in Table 2 are relevant only to CICS or IMS application development. See “Getting Started in IMS” on page 49 or “Getting Started in CICS” on page 121 for more details.
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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Define public IDL interfaces to the objects required in your system. See “Defining IDL Interfaces” on page 23.</td>
</tr>
<tr>
<td>2</td>
<td>Use the ORXCOPY utility to copy your IDL files to z/OS (if necessary). See “ORXCOPY Utility” on page 523.</td>
</tr>
<tr>
<td>3</td>
<td>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 24.</td>
</tr>
</tbody>
</table>
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.DEMOSIDL(SIMPLE):

```idl
// 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.

**Note:** If your IDL files are not already contained in z/OS data sets, you must copy them to z/OS before you proceed. You can use the ORXCOPY utility to do this. If necessary, see “ORXCOPY Utility” on page 523 for more details.

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 331 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.JCLLIB(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>
<thead>
<tr>
<th>Member</th>
<th>JCL Keyword Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idlimembername1</td>
<td>IMPL</td>
<td>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.</td>
</tr>
</tbody>
</table>
Developing the Application Interfaces

Table 3: Generated Server Source Code Members

<table>
<thead>
<tr>
<th>Member</th>
<th>JCL Keyword Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idlmembernameV</td>
<td>IMPL</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 293 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

<table>
<thead>
<tr>
<th>Copybook</th>
<th>JCL Keyword Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idlmembernameE</td>
<td>COPYLIB</td>
<td>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.</td>
</tr>
</tbody>
</table>
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` include member.

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` include member. This member is automatically included in the `idlmembername` include member.

This include member contains the based structure declarations that are used in the `idlmembername` include member. This member is automatically included in the `idlmembername` include member.

<table>
<thead>
<tr>
<th>Copybook</th>
<th>JCL Keyword Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>idlmembername</code></td>
<td>COPYLIB</td>
<td>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 <code>idlmembername</code> include member.</td>
</tr>
<tr>
<td><code>idlmembernameM</code></td>
<td>COPYLIB</td>
<td>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 <code>idlmembernameM</code> include member. This member is automatically included in the <code>idlmembername</code> include member.</td>
</tr>
<tr>
<td><code>idlmembernameT</code></td>
<td>COPYLIB</td>
<td>This include member contains the based structure declarations that are used in the <code>idlmembernameT</code> include member. This member is automatically included in the <code>idlmembernameM</code> include member.</td>
</tr>
</tbody>
</table>
Developing the Application Interfaces

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 235 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.

Table 4: Generated PL/I Include Members

<table>
<thead>
<tr>
<th>Copybook</th>
<th>JCL Keyword Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idlmembernameX</td>
<td>COPYLIB</td>
<td>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 source code member.</td>
</tr>
<tr>
<td>idlmembernameD</td>
<td>COPYLIB</td>
<td>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 source code member.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How IDL maps to PL/I include members</th>
<th>Table 4: Generated PL/I Include Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 235 for details of how IDL types map to PL/I.</td>
<td></td>
</tr>
<tr>
<td>Attributes map to two operations (get and set), and readonly attributes map to a single get operation.</td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“Writing the Server Implementation” on page 30</td>
</tr>
<tr>
<td>2</td>
<td>“Writing the Server Mainline” on page 33</td>
</tr>
<tr>
<td>3</td>
<td>“Building the Server” on page 36.</td>
</tr>
</tbody>
</table>
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*/
DISPTCH: ENTRY;

dcl (addr, low, sysnull) builtin;
%include CORBA;
%include CHKERRS;
%include SIMPLM;
%include DISPINIT;
/* ================ Start of global user code ================ */
/* ================ End of global user code ================ */
/* ------------------------------------------------------------------
/* Dispatcher : select(operation) */
/* ------------------------------------------------------------------*/
%include SIMPLED;
/* Interface: */
/* Simple/SimpleObject */
/* */
```
The SIMPLEx 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 SIMPLEx include member contains a select statement that determines which procedure within SIMPLEx is to be called, given the operation name and interface name passed to SIMPLEx. It calls PODGET

Example 1: The SIMPLEx Demonstration Module (Sheet 2 of 2)
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(SIMPLE)
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);
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 */
6     open file(IORFILE);
     call obj2str(Simple_SimpleObject_obj, iorrec_ptr);
     if check_errors('obj2str') ^= completion_status_yes then return;
     put skip list('Writing out the object reference');
8     call podrun;
     if check_errors('podrun') ^= completion_status_yes then return;
     call objrel(Simple_SimpleObject_obj);
     if check_errors('objrel') ^= completion_status_yes then return;
     free pod_status_information;
     close file(IORFILE);
8     put skip;
   7     call podrun;
     if check_errors('podrun') ^= completion_status_yes then return;
     free pod_status_information;
   6     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;
   5     call objnew(server_name, Simple_SimpleObject_intf,
             Simple_SimpleObject_objid, Simple_SimpleObject_obj);
     if check_errors('objnew') ^= completion_status_yes then return;
     put skip list('Creating the simple_persistent object');
   4     call podreg(addr(Simple_SimpleObject_interface));
     if check_errors('podreg') ^= completion_status_yes then return;
     /* Register interface : Simple/SimpleObject */
3     call podsrvr(server_name, server_name_len);
     if check_errors('podsrvr') ^= completion_status_yes then return;
2     call orbargs(arg_list, arg_list_len, orb_name, orb_name_len);
     if check_errors('orbargs') ^= completion_status_yes then return;
```
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. `PODCREG` 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

<table>
<thead>
<tr>
<th>Location of the JCL</th>
<th>Sample JCL used to compile and link the batch server mainline and server implementation is in <code>orbixhlq.DEMOS.PLI.BLD..JCLLIB(SIMPLESB)</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resulting load module</td>
<td>When this JCL has successfully executed, it results in a load module that is contained in <code>orbixhlq.DEMOS.PLI.LOADLIB(SIMPLESV)</code>.</td>
</tr>
<tr>
<td>Server programming restrictions</td>
<td>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: <em>IBM PL/I for MVS &amp; VM Language Reference Release 1.1: SC26-3114</em> 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.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“Writing the Client” on page 38.</td>
</tr>
<tr>
<td>2</td>
<td>“Building the Client” on page 41.</td>
</tr>
</tbody>
</table>
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)

4 /* Register the SimpleObject interface with the PL/I runtime */
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 */
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...’);
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 */
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

<table>
<thead>
<tr>
<th>Location of the JCL</th>
<th>Sample JCL used to compile and link the client can be found in the third step of <code>orbixhlq.DEMOS.PLI.BLD.JCLLIB(SIMPLECB)</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resulting load module</td>
<td>When the JCL has successfully executed, it results in a load module that is contained in <code>orbixhlq.DEMOS.PLI.LOADLIB(SIMPLECL)</code>.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“Starting the Orbix Locator Daemon” on page 43 (if it has not already been started).</td>
</tr>
<tr>
<td>2</td>
<td>“Starting the Orbix Node Daemon” on page 44 (if it has not already been started).</td>
</tr>
<tr>
<td>3</td>
<td>“Running the Server and Client” on page 45.</td>
</tr>
</tbody>
</table>
## 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.JCLLIB(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.JCLLIB(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 Application

### Running the Server and Client

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

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

**Note:** You can use the z/OS `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.JCLLIB(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 z/OS environment. This is shown for the purposes of example and is not meant to reflect a real-world scenario requiring the Orbix Mainframe.

![Address Space Layout for an Orbix PL/I Application](image)

**Figure 3: Address Space Layout for an Orbix PL/I Application**
CHAPTER 2 | Getting Started in Batch

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 35 and “API Reference” on page 377.
- The APIs called by the server implementation, see “Explanation of the SIMPLEI module” on page 31 and “API Reference” on page 377.

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 40 and “API Reference” on page 377.
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. Additionally, this chapter shows how to develop an IMS client that supports two-phase commit transactions.

In this chapter

This chapter discusses the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>51</td>
</tr>
<tr>
<td>Developing the Application Interfaces</td>
<td>59</td>
</tr>
<tr>
<td>Developing the IMS Server</td>
<td>71</td>
</tr>
<tr>
<td>Developing the IMS Client</td>
<td>83</td>
</tr>
<tr>
<td>Developing the IMS Two-Phase Commit Client</td>
<td>93</td>
</tr>
<tr>
<td>Running the Demonstrations</td>
<td>114</td>
</tr>
</tbody>
</table>
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 the following Orbix PL/I applications:

- IMS server
- IMS client
- IMS two-phase commit client

It also introduces the following PL/I demonstrations that are supplied with your Orbix Mainframe installation, and outlines where you can find the various source code and JCL elements for them:

- SIMPLE IMS server
- SIMPLE IMS client
- DATACL IMS two-phase commit client

Steps to create an application

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

1. "Developing the Application Interfaces" on page 59.
2. "Developing the IMS Server" on page 71.
3. "Developing the IMS Client" on page 83.
4. "Developing the IMS Two-Phase Commit Client" on page 93.

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. Additionally, this chapter describes how to develop an Orbix PL/I two-phase commit IMS client, and run it against two C++ servers. The supplied demonstrations do not reflect real-world scenarios requiring 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 object. 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.

The demonstration IMS two-phase commit client

The Orbix PL/I two-phase commit client developed in this chapter runs in an IMS region. It uses the clearly defined object interface, Data, to access and update data from two Orbix C++ batch servers. When the client invokes a remote operation, a request message is sent from the client to one of the servers 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 servers.

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 4)

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orbixhlq.DEMOS.IDL(SIMPLE)</td>
<td>This is the supplied IDL for the simple IMS client and server.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.IDL(DATA)</td>
<td>This is the supplied IDL for the IMS two-phase commit client.</td>
</tr>
</tbody>
</table>
This is the source code for the IMS server mainline module, which is generated when you run the JCL in `orbixhlq.DEOMS.IMS.PLI.BLD.JCLLIB(SIMPLIDL)`. (The IMS server mainline code is not shipped with the product. You must run the `SIMPLIDL` JCL to generate it.)

This is the source code for the IMS server implementation module.

This is the source code for the IMS simple client module.

This is the source code for the IMS two-phase commit client module.

This JCL runs the Orbix IDL compiler. See “Orbix IDL Compiler” on page 62 for more details of this JCL and how to use it.

This JCL compiles and links the IMS server mainline and IMS server implementation modules to create the SIMPLE server program.

This JCL compiles the IMS simple client module to create the SIMPLE client program.

This JCL compiles the IMS two-phase commit client module.

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>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>orbixhlq.DEOMS.IMS.PLI.SRC (SIMPLESV)</code></td>
<td>This is the source code for the IMS server mainline module, which is generated when you run the JCL in <code>orbixhlq.DEOMS.IMS.PLI.BLD.JCLLIB(SIMPLIDL)</code>. (The IMS server mainline code is not shipped with the product. You must run the <code>SIMPLIDL</code> JCL to generate it.)</td>
</tr>
<tr>
<td><code>orbixhlq.DEOMS.IMS.PLI.SRC (SIMPLES)</code></td>
<td>This is the source code for the IMS server implementation module.</td>
</tr>
<tr>
<td><code>orbixhlq.DEOMS.IMS.PLI.SRC (SIMPLECL)</code></td>
<td>This is the source code for the IMS simple client module.</td>
</tr>
<tr>
<td><code>orbixhlq.DEOMS.IMS.PLI.SRC (DATACL)</code></td>
<td>This is the source code for the IMS two-phase commit client module.</td>
</tr>
<tr>
<td><code>orbixhlq.DEOMS.IMS.PLI.BLD.JCLLIB (SIMPLIDL)</code></td>
<td>This JCL runs the Orbix IDL compiler. See “Orbix IDL Compiler” on page 62 for more details of this JCL and how to use it.</td>
</tr>
<tr>
<td><code>orbixhlq.DEOMS.IMS.PLI.BLD.JCLLIB (SIMPLES)</code></td>
<td>This JCL compiles and links the IMS server mainline and IMS server implementation modules to create the SIMPLE server program.</td>
</tr>
<tr>
<td><code>orbixhlq.DEOMS.IMS.PLI.BLD.JCLLIB (SIMPLECB)</code></td>
<td>This JCL compiles the IMS simple client module to create the SIMPLE client program.</td>
</tr>
<tr>
<td><code>orbixhlq.DEOMS.IMS.PLI.BLD.JCLLIB (DATACB)</code></td>
<td>This JCL compiles the IMS two-phase commit client module.</td>
</tr>
<tr>
<td><code>orbixhlq.DEOMS.IMS.PLI.BLD.JCLLIB (SIMPLIOR)</code></td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table 5: Supplied Code and JCL (Sheet 3 of 4)

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
</table>
| `orbixhlq.DEMOS.IMS.PLI.BLD.JCLLIB (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 481 for more details of the various forms of corbaloc URL strings and the ways you can use them. |
| `orbixhlq.DEMOS.IMS.PLI.BLD.JCLLIB (DATAIORS)` | This JCL adds the following configuration entries to the configuration member:  
`initial_references:DataObjectA:reference="IOR...";`  
`initial_references:DataObjectB:reference="IOR...";`  
These configuration entries specify the IORs that the IMS two-phase commit client uses to contact the C++ batch servers. The IORs that are set as the values for these configuration entries are the IORs that are published in `orbixhlq.DEMOS.IORS (DATAA)` and `orbixhlq.DEMOS.IORS (DATAB)` when you run the C++ batch servers.  
The object references for the servers are represented to the demonstration IMS two-phase commit client as corbaloc URL strings in the form `corbaloc:rir:/DataObjectA` and `corbaloc:rir:/DataObjectB`. This form of corbaloc URL string requires the use of the `initial_references:DataObjectA:reference="IOR..."` and `initial_references:DataObjectA:reference="IOR..."` configuration items. |
<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orbixhlq.JCLLIB(IMSCA)</td>
<td>This JCL runs the IMS client adapter.</td>
</tr>
<tr>
<td>orbixhlq.JCLLIB(IMSA)</td>
<td>This JCL runs the IMS server adapter.</td>
</tr>
<tr>
<td>orbixhlq.DMOS.CPP.BLD.JCLLIB (DATASV)</td>
<td>This JCL builds the C++ servers for the IMS two-phase commit client.</td>
</tr>
<tr>
<td>orbixhlq.DMOS.CPP.RUN.JCLLIB (DATAA)</td>
<td>This JCL runs the C++ server 'A' for the IMS two-phase commit client.</td>
</tr>
<tr>
<td>orbixhlq.DMOS.CPP.RUN.JCLLIB (DATAB)</td>
<td>This JCL runs the C++ server 'B' for the IMS two-phase commit client.</td>
</tr>
<tr>
<td>orbixhlq.DMOS.CPP.GEN</td>
<td>This PDS contains generated stub code for the C++ servers.</td>
</tr>
<tr>
<td>orbixhlq.DMOS.CPP.H</td>
<td>This PDS contains C++ header files.</td>
</tr>
<tr>
<td>orbixhlq.DMOS.CPP.HH</td>
<td>This PDS contains IDL generated header files.</td>
</tr>
<tr>
<td>orbixhlq.DMOS.CPP.LOADLIB</td>
<td>This PDS contains the C++ server module for the two-phase commit IMS client.</td>
</tr>
<tr>
<td>orbixhlq.DMOS.CPP.SRC</td>
<td>This PDS contains the C++ server module source code for the two-phase commit IMS client.</td>
</tr>
<tr>
<td>orbixhlq.DMOS.CPP.TWOPCA</td>
<td>This PDS contains the data store for the two-phase commit C++ server 'A'.</td>
</tr>
<tr>
<td>orbixhlq.DMOS.CPP.TWOPCB</td>
<td>This PDS contains the data store for the two-phase commit C++ server 'B'.</td>
</tr>
</tbody>
</table>
**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)**

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>orbixhlq.INCLUDE.PLINCL(CHKCLIMS)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>orbixhlq.INCLUDE.PLINCL(CHKErrs)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>orbixhlq.INCLUDE.PLINCL(CORBA)</code></td>
<td>This is relevant to both IMS clients and servers. It contains common PL/I runtime variables. It includes the <code>CORBACOM</code> include member by default. It also includes the <code>CORBASV</code> include member, if the client module contains the line <code>&amp;client_only='yes'</code>.</td>
</tr>
<tr>
<td><code>orbixhlq.INCLUDE.PLINCL(CORBACOM)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>orbixhlq.INCLUDE.PLINCL(CORBASV)</code></td>
<td>This is relevant to IMS servers. It contains PL/I runtime function definitions that can be used by servers.</td>
</tr>
<tr>
<td><code>orbixhlq.INCLUDE.PLINCL(DISFINIT)</code></td>
<td>This is relevant to IMS servers only. It retrieves the current request information into the <code>REQINFO</code> structure via <code>PODREQ</code>. From <code>REQINFO</code> the operation to be performed by the server is retrieved via a call to <code>STRGET</code>.</td>
</tr>
<tr>
<td><code>orbixhlq.INCLUDE.PLINCL(DLIDATA)</code></td>
<td>This is relevant to IMS clients only. It contains structures to facilitate reading from and writing to the IMS message queue via <code>iopcb_ptr</code>. It contains a PL/I function called <code>write_dc_text</code> that facilitates writing messages to the IMS output message queue. It does this by using the supplied IBM routine (interface) <code>PLITDLI</code> to make an IMS DC (data communications) call that specifies the common IMS function command <code>ISRT</code> (insert). The <code>DLIDATA</code> member contains all the declarations needed for the supplied PL/I client demonstration in IMS.</td>
</tr>
</tbody>
</table>
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.

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.

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.

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 481 for more details.

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.

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orbixhlq.INCLUDE.PLINCL(EXCNAME)</td>
<td>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.</td>
</tr>
<tr>
<td>orbixhlq.INCLUDE.PLINCL(GETUNIQ)</td>
<td>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.</td>
</tr>
<tr>
<td>orbixhlq.INCLUDE.PLINCL(IMSPCB)</td>
<td>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.</td>
</tr>
<tr>
<td>orbixhlq.INCLUDE.PLINCL(URLSTR)</td>
<td>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 481 for more details.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.IMS.PLI.PLINCL</td>
<td>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.</td>
</tr>
</tbody>
</table>
Checking JCL components

When creating the IMS simple client or server, or the IMS two-phase commit client, 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.

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orbixhlq.DEMOS.IMS.MFAMAP</td>
<td>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 qualified 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.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.TYPEINFO</td>
<td>This PDS is relevant to IMS servers only. It is empty at installation time. It is used to store the type information that is generated when you run the JCL to run the Orbix IDL compiler for the supplied demonstrations. The contents of the type information member describe the contents of the given IDL file from which it was generated.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
</table>
| 1    | Define public IDL interfaces to the objects required in your system.  
See “Defining IDL Interfaces” on page 60. |
| 2    | Use the ORXCOPY utility to copy your IDL files to z/OS (if necessary).  
See “ORXCOPY Utility” on page 523. |
| 3    | Run the Orbix IDL compiler to generate PL/I include members, server source, and server mapping member.  
See “Orbix IDL Compiler” on page 62. |
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
// 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.PLL.LD.JCLLIB(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:** If your IDL files are not already contained in z/OS data sets, you must copy them to z/OS before you proceed. You can use the ORXCOPY utility to do this. If necessary, see “ORXCOPY Utility” on page 523 for more details.

**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 293 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:

```plaintext
//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.PROCLIB)
// INCLUDE MEMBER=(ORXVARS)
```
Developing the Application Interfaces

Explanation of the SIMPLIDL JCL

In the preceding JCL example, the lines IDLPARM='-pli:-V' and IDLPARM='-pli:-TIMS -mfa:-tSIMPLESV:-inf' are mutually exclusive. The line IDLPARM='-pli:-TIMS -mfa:-tSIMPLESV:-inf' 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:-ttran_name arguments.
- Type information for the SIMPLE IDL member via the -inf sub-argument to the -mfa argument.

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, an IMS server adapter mapping member, and type information for the SIMPLE IDL member.

Alternatively, if you choose to comment out the line that has the -pli:-TIMS
-mfa:-tSIMPLESV:-inf arguments, the IDL compiler only generates PL/I include members.
See “Orbix IDL Compiler” on page 293 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.JCLLIB(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 235 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.
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)**

<table>
<thead>
<tr>
<th>Copybook</th>
<th>JCL Keyword Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>idlmembernameD</code></td>
<td>COPYLIB</td>
<td>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.</td>
</tr>
<tr>
<td><code>idlmembernameL</code></td>
<td>COPYLIB</td>
<td>This include member contains structures and procedures used by the PL/I runtime to read and store data into the operation parameters.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This member is automatically included in the <code>idlmembernameX</code> include member.</td>
</tr>
<tr>
<td><code>idlmembernameM</code></td>
<td>COPYLIB</td>
<td>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 <code>idlmembernameT</code> include member.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This member is automatically included in the <code>idlmembernameI</code> include member.</td>
</tr>
</tbody>
</table>
Developing the Application Interfaces

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

<table>
<thead>
<tr>
<th>Copybook</th>
<th>JCL Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idlmembernameT</td>
<td>COPYLIB</td>
<td>This include member contains the based structure declarations that are used in the \textit{idlmembername}\text{M} include member. This member is automatically included in the \textit{idlmembername}\text{M} include member.</td>
</tr>
<tr>
<td>idlmembernameX</td>
<td>COPYLIB</td>
<td>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 \textit{idlmembername}\text{V} source code member.</td>
</tr>
<tr>
<td>idlmembernameD</td>
<td>COPYLIB</td>
<td>This include member contains a select statement for calling the correct procedure for the requested operation. This include member is automatically included in the \textit{idlmembername}\text{I} source code member.</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Member</th>
<th>JCL Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idlmembernameI</td>
<td>IMPL</td>
<td>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.</td>
</tr>
<tr>
<td>idlmembernameV</td>
<td>IMPL</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 293 for more details of the IDL compiler arguments used to generate, and prevent generation of, IMS server source code.
Developing the Application Interfaces

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

<table>
<thead>
<tr>
<th>Copybook</th>
<th>JCL Keyword Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idlmembernameA</td>
<td>IDLMFA</td>
<td>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.</td>
</tr>
</tbody>
</table>

Table 10 shows the type information member that the Orbix IDL compiler generates, based on the defined IDL.

Table 10: Generated Type Information Member

<table>
<thead>
<tr>
<th>Copybook</th>
<th>JCL Keyword Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idlmembernameB</td>
<td>IDLTYPEI</td>
<td>Type information describing the operation signatures of the interface whose IDL it was generated from.</td>
</tr>
</tbody>
</table>

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.MFANAP(SIMPLEA)
• orbixhlq.DEMOS.TYPEINFO(SIMPLEB)

**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.JCLLIB(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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“Writing the Server Implementation” on page 72.</td>
</tr>
<tr>
<td>2</td>
<td>“Writing the Server Mainline” on page 75.</td>
</tr>
<tr>
<td>3</td>
<td>“Building the Server” on page 78.</td>
</tr>
<tr>
<td>4</td>
<td>“Preparing the Server to Run in IMS” on page 79.</td>
</tr>
</tbody>
</table>
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 includemember. For the purposes of this example, you must write a PL/I procedure that implements each operation in the SIMPLET includemember. When you specify the -S and -TIMS arguments with the Orbix IDL compiler, it generates a skeleton server implementation module, in this case called SIMPLEx, which is a useful starting point.

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

Example of the IMS SIMPLEx module

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

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

```pli
SIMPLEX: PROC;

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

dcl (addr,low,sysnull) builtin;

#include CORBA;
#include CHKERRS;
#include DLIDATA;
#include IMSPCB;
#include SIMPLEM;
#include DISPINIT;

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

/* Dispatcher : select(operation) */
/* */
/* */
/* */
/* */
```
Developing the IMS Server

Table 11: The SIMPLEI Demonstration Module  (Sheet 2 of 2)

3 %include SIMPLED;
   /*------------------------------------------------------------------------*/
   /* Interface:                                                           */
   /*   Simple/SimpleObject                                                */
   /*                                                             */
   /*   Mapped name:                                                       */
   /*   Simple_SimpleObject                                                */
   /*                                                             */
   /*   Inherists interfaces:                                             */
   /*     (none)                                                          */
   /*------------------------------------------------------------------------*/
   /* Operation:          call_me                                          */
   /* Mapped name:         call_me                                          */
   /* Arguments:           None                                            */
   /* Returns:             void                                            */
   /*------------------------------------------------------------------------*/

proc_Simple_SimpleObject_c_c904: PROC(p_args);
   dcl p_args        ptr;

5 dcl l_args           aligned based(p_args)
                        like Simple_SimpleObject_c_ba77_type;

6 /* ============ Start of operation specific code ============= */
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.

Note: Although not used by the SIMPLE demonstration, DLIDATA and IMSPCB provide a means of writing data to the IMS console and access to the IMS PCB pointers read in and stored by the SIMPLEV member.
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.DENOS.IMS.PLI.SRC(SIMPLEI).
Developing the IMS Server

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

```plaintext
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 SETUPSV;
   %include CORBA;
   %include CHKERRS;
   %include IMSPCB;
   %include SIMPLET;
   %include SIMPLEX;
```
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.
Developing the IMS Server

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 SIMPLESV server mainline module in `orbixhlq.DEMOS.IMS.PLL.SRC(SIMPLEV)` after you have run `orbixhlq.DEMOS.IMS.PLL.BLD.JCLLIB(SIMPLIDL)` to run the Orbix IDL compiler.
### Building the Server

<table>
<thead>
<tr>
<th>Location of the JCL</th>
<th>Sample JCL used to compile and link the IMS server mainline and server implementation is in <code>orbixhlq.DEMOS.IMS.PLI.BLD.JCLLIB(SIMPLESB)</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resulting load module</td>
<td>When this JCL has successfully executed, it results in a load module that is contained in <code>orbixhlq.DEMOS.IMS.PLI.LOADLIB(SIMPLESV)</code></td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Define a transaction definition for IMS.</td>
</tr>
<tr>
<td>2</td>
<td>Provide the IMS server load module to an IMS region.</td>
</tr>
<tr>
<td>3</td>
<td>Generate mapping member entries for the IMS server adapter.</td>
</tr>
<tr>
<td>4</td>
<td>Add the interface’s operation signatures to the type information repository, stored in the TYPEINFO PDS.</td>
</tr>
<tr>
<td>5</td>
<td>Obtain the IOR for use by the client program.</td>
</tr>
</tbody>
</table>

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
PGMTRYPE=(TP, 2), x
SCHDTYP=PARALLEL
TRANSACT   CODE=SIMPLESV, x
            EDIT=(ULC)
```

Step 2—Providing load module to IMS region

Ensure that the `orbixhq.DEMOS.IMS.PLI.LOADLIB` 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 `orbxhlq.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 `orbxhlq.DEMOS.IMS.PLI.BLD.JCLLIB(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 operation signatures to type_info store

The IMS server adapter needs to be able to obtain operation signatures for the PL/I server. For the purposes of this demonstration, the TYPEINFO PDS is used to store 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. This information is generated by supplying the `-mfa:-inf` option to the IDL compiler, for example, as used in the SIMPLIDL JCL used to generate the source and include members for this demonstration.

Note: An IDL interface only needs to be added to the type information store once.
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 type_info store contains the relevant operation signatures (or, if using the IFR, the IFR is running and contains the relevant IDL). See “Step 4—Adding operation signatures to type_info store” on page 80 for details of how to populate the type_info store.

• The IMS server adapter mapping member contains the relevant mapping entries. For the purposes of this example, ensure that the orbixhlq.DEmons.IMS.MFAMAP(SIMPLEA) mapping member is being used. See the IMS Adapters Administrator’s Guide for details about IMS server adapter mapping members.

• The IMS server adapter is running. See the IMS Adapters Administrator’s Guide for more details of how to start the IMS server adapter, using the supplied JCL in orbixhlq.JCLIB(IMSA) JCL.

Note: An alternative to using type information files is to use the Interface Repository (IFR). This is an alternative method of allowing the IMS server adapter to retrieve IDL type information. If you are using the IFR, you must ensure that the relevant IDL for the server has been added to the IFR (that is, registered with it) before the IMS server adapter is started.

To add IDL to the IFR, first ensure the IFR is running. You can use the JCL in orbixhlq.JCL(IFR) to start it. Then, in the JCL that you use to run the Orbix IDL compiler, add the line // IDLPARM=-R’ to register the IDL. In this case, ensure that all other // IDLPARM lines are commented out as follows: //* IDLPARM...
Now submit `orbixhlq.DEMOS.IMS.PLI.BLD.JCLLIB(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.PROCLIB)
//* 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.

// 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=&ORBIXCFG(&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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“Writing the Client” on page 84.</td>
</tr>
<tr>
<td>2</td>
<td>“Building the Client” on page 88.</td>
</tr>
<tr>
<td>3</td>
<td>“Preparing the Client to Run in IMS” on page 89.</td>
</tr>
</tbody>
</table>
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 13: The SIMPLEC Demonstration Module (Sheet 1 of 3)

```plaintext
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;

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

%include CORBA;
%include IMSPCB;
%include DLIDATA;
%include GETUNIQ;
%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);
```
developing the ims client

Table 13: The SIMPLEC Demonstration Module (Sheet 2 of 3)

```plaintext
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 */
7 so we can make calls to the server */
call strset(simple_url_ptr,
    simple_url,
    length(simple_url));
if check_errors('strset') ^= completion_status_yes then
    return;
/* Now we are ready to start making server requests */
call str2obj(simple_url_ptr,Simple_SimpleObject_obj);
if check_errors('str2obj') ^= completion_status_yes then
    return;
/* 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. */
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;
```

85
CHAPTER 3 | Getting Started in IMS

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 481 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 `orbix.hlq.Demos.IMS.PLI.BLD.JCLLIB(SIMPLEC)`.

Resulting load module

When the JCL has successfully executed, it results in a load module that is contained in `orbix.hlq.Demos.IMS.PLI.LOADLIB(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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Define an APPC transaction definition for IMS.</td>
</tr>
<tr>
<td>2</td>
<td>Provide the IMS client load module to an IMS region.</td>
</tr>
<tr>
<td>3</td>
<td>Start the locator and node daemon on the server host.</td>
</tr>
<tr>
<td>4</td>
<td>Add the interface's operation signatures to the type information repository.</td>
</tr>
<tr>
<td>5</td>
<td>Start the batch server.</td>
</tr>
<tr>
<td>6</td>
<td>Customize the batch server IOR.</td>
</tr>
<tr>
<td>7</td>
<td>Configure and run the client adapter.</td>
</tr>
</tbody>
</table>

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.LOADLIB` 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 and node daemon on server host

This step assumes 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.JCLLIB(LOCATOR)`.
2. Start the node daemon by submitting `orbixhlq.JCLLIB(NODEDAEM)`.

See “Running the Server and Client” on page 45 for more details of running the locator and node daemon on the batch server host.

Step 4—Add operation signatures to type_info store

The client adapter needs to be able to know what data types it can expect to marshal from the IMS transaction, and what data types it should expect back from the batch server. This can be done by creating a type information file by running the IDL compiler with the `-mfa:-inf` flag, which is included in `orbixhlq.DEMOS.IMS.PLI.BLD.JCLLIB(SIMPLIDL)`. The type information file contains descriptions of the interface’s operation signatures (that is, information about the type and direction of the operation parameters, the number of parameters, and whether or not an operation has a return type).

Before the client adapter is run, the `TYPEINFO` DD card needs to be updated to the location of the `TYPEINFO` PDS (for the purposes of this example, it should be updated to `orbixhlq.DEMOS.TYPEINFO`).

**Note:** An IDL interface only needs to be added to the type information store once.
Step 5—Start batch server

This step assumes 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.JCLLIB(SIMPLESV)
```

See “Running the Server and Client” on page 45 for more details of running the locator and node daemon on the batch server host.

Note: An alternative to using type information files is to use the Interface Repository (IFR). This is an alternative method of allowing the client adapter to obtain information about relevant data types. If you are using the IFR, you must ensure that the relevant IDL for the server has been added to the IFR (that is, registered with it) before the client adapter is started.

To add IDL to the IFR, first ensure the IFR is running. You can use the JCL in `orbixhlq.JCL(IFR)` to start it. Then, in the JCL that you use to run the Orbix IDL compiler, add the line `// IDLPARM='-R'` to register the IDL. In this case, ensure that all other `// IDLPARM` lines are commented out as follows: `//* IDLPARM...`

Step 6—Customize batch server

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.JCLLIB(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 481 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 (the 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 `orbixhlq.JCLLIB(IMSCA)`:  

| Note: | See “Running the Demonstrations” on page 114 for details of how to run the sample demonstration. |
Developing the IMS Two-Phase Commit Client

**Overview**

This section describes the steps you must follow to develop the IMS two-phase commit client executable for your application. The IMS two-phase commit client developed in this example will connect to two demonstration C++ batch servers.

**Steps to develop the client**

The steps to develop and run the client application are:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;Writing the Client&quot; on page 94.</td>
</tr>
<tr>
<td>2</td>
<td>&quot;Building the Client&quot; on page 109.</td>
</tr>
<tr>
<td>3</td>
<td>&quot;Building the Servers&quot; on page 110.</td>
</tr>
<tr>
<td>4</td>
<td>&quot;Preparing the Client to Run in IMS&quot; on page 111.</td>
</tr>
</tbody>
</table>
Writing the Client

The client program

The next step is to write the IMS client transaction. This example uses the supplied DATAC client demonstration.

IMS transaction design

An IMS transaction that uses two-phase commit can be broken down as follows:

- Operations that do not require two-phase commit.
- Operations that require two-phase commit.

Read-only operations to local databases or remote servers do not require two-phase commit processing. These operations should be performed first in the IMS transaction ahead of the two-phase commit operations. The rationale behind this is that if operations not requiring two-phase commit processing fail, it might be pointless to perform operations that do require two-phase commit processing.

Overview of IMS transaction layout

Figure 4 provides an overview of IMS transaction layout.

![Figure 4: Overview of IMS Transaction Layout](image-url)
Designing an IMS two-phase commit transaction

When designing an IMS two-phase commit transaction, structure the transaction as follows:

1. Begin the IMS transaction by performing standard Orbix Mainframe IMS client initialization.
2. Issue an initial IMS Get Unique call.
3. Perform the following loop until the IMS status code indicates that there are no more segments:
   • Perform operations that do not require two-phase commit. If any of the operations fail, skip the two-phase commit processing.
   • Call `PODTXNB` to indicate the start of two-phase commit processing.
   • Call `PODEXEC` (perhaps multiple times) to send an update to a remote server. If any of the calls fail, call rollback and skip any updates to local resources.
   • Make updates to local resources, such as updating a local database. If any of the local updates fail, call rollback.
   • Call `PODTXNE` to indicate the end of the two-phase commit work.
   • Perform any post two-phase commit work, such as sending a message back to the user.
   • Issue another Get Unique call.
4. End loop.

Commit or rollback scenarios

When an IMS transaction makes updates to resources (that is, local databases or remote CORBA servers) via the client adapter, the updates are not made permanent until the two-phase commit has been successfully processed. The trigger for starting the two-phase commit is when the IMS transaction finishes its processing. The transaction does not immediately end. Instead, it waits for the results of two-phase commit to decide whether it should commit or roll back its updates to local resources.

The client adapter sends a "prepare" message to each remote server that has been updated from the IMS transaction. Each server returns a vote to the client adapter. A vote of "commit" indicates the remote server is willing to commit its updates. A vote of "rollback" indicates the remote server has a problem and that it wants to roll back the update.
The various scenarios that might arise are as follows:

- **Successful two-phase commit**
  If all returned votes are "commit", the client adapter calls the IBM API `SRRCMIT`, to inform IMS that all remote servers are willing to commit their updates. If the return code from `SRRCMIT` is 0, the client adapter sends a "commit" message to each remote server. Two-phase commit processing is then completed and all resources are updated.

- **Rollback two-phase commit—Scenario 1**
  If the client adapter receives at least one returned vote of "rollback", all updates should be rolled back. The client adapter calls the IBM API `SRRBACK`, to inform IMS that there are problems. This causes the IMS transaction to abend with a U0711 code to roll back any local updates.

- **Rollback two-phase commit—Scenario 2**
  If all returned votes are "commit", the client adapter calls the IBM API `SRRCMIT`, to inform IMS that all remote servers are willing to commit their updates. If the return code from `SRRCMIT` is not 0, the client adapter sends a "rollback" message to each server. In this case, this means that a resource other than the remote servers has voted "rollback".

- **Rollback two-phase commit—Scenario 3**
  If the IMS transaction makes an update to a remote server, and the update fails (because, for example, the server is not running), the transaction calls "rollback" to undo any updates. The client adapter receives the rollback signal and sends a "rollback" message to each server.
Example of the DATAC module

The following is an example of the IMS DATAC module:

Example 4: The DATAC Demonstration Module (Sheet 1 of 10)

```plaintext
DATAC: 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,string,substr,sysnull,length) builtin;

dcl arg_list                   char(40) init('');

dcl arg_list_len               fixed bin(31) init(38);

dcl orb_name                   char(9) init('twopc_orb');

dcl orb_name_len               fixed bin(31) init(9);

dcl sysprint                   file stream output;

dcl data_urlA                  char(26) init('corbaloc:rir:/DataObjectA ');

dcl data_urlB                  char(26) init('corbaloc:rir:/DataObjectB ');

dcl data_url_ptr               ptr init(sysnull());

dcl DataObject_objA            ptr;

dcl DataObject_objB            ptr;

dcl read_result_A              fixed bin(31) init(0);

dcl update_result_A            fixed bin(31) init(0);

dcl read_result_B              fixed bin(31) init(0);

dcl update_result_B            fixed bin(31) init(0);

dcl good_result                fixed bin(31) init(1);

dcl 1 dc_text_area,           fixed bin(31),
    3 msg_length             fixed bin(31),
    3 dc_text_msg,          fixed bin(31),
    5 header               char(42),
    5 result               char(08);

%include CORBA;
%include IMSCPB;
%include DLIDATA;
%include CHKCLIMS;
%include DATAM;
%include DATAX;
```
Example 4: The DATAC Demonstration Module (Sheet 2 of 10)

```plaintext
call Initialize;

do until(io_pcb.status_code ^= '  ');
    call Process_transaction;
end;

call Terminate;
```

```
Initialize: PROC;

call GET_UNIQ;
```

```
/* Initialize the PL/I runtime status information block */
Example 4: The DATAC Demonstration Module (Sheet 3 of 10)

```c
alloc pod_status_information set(pod_status_ptr);
call podstat(pod_status_ptr);

/* Initialize our ORB */
put skip list('Initializing the ORB');
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 */
put skip list('Registering the Interface');
call podreg(addr(Data_interface_interface));
if check_errors('podreg') ^= completion_status_yes then return;

/* Set the pointer to the urlA string. */
call strset(data_url_ptr,
    data_urlA,
    length(data_urlA));
if check_errors('strset') ^= completion_status_yes then return;

/* Obtain object A reference from the url. */
call str2obj(data_url_ptr,DataObject_objA);
if check_errors('str2obj') ^= completion_status_yes then return;

/* Releasing the memory. */
call strfree(data_url_ptr);
if check_errors('strfree') ^= completion_status_yes then return;

/* Set the pointer to the urlB string. */
call strset(data_url_ptr,
    data_urlB,
    length(data_urlB));
if check_errors('strset') ^= completion_status_yes then return;

/* Obtain object B reference from the url. */
call str2obj(data_url_ptr,DataObject_objB);
if check_errors('str2obj') ^= completion_status_yes then return;

/* Releasing the memory. */
```
Example 4: The DATAC Demonstration Module (Sheet 4 of 10)

call strfree(data_url_ptr);
if check_errors('strfree') ^= completion_status_yes then
  return;
END Initialize;

/*************************************************************/
/*                                                           */
/* Process_transaction                                       */
/*                                                           */
/* Begin a two-phase commit transaction by calling podtxnb.   */
/* Read a value from "server A". Add 1 to the value and      */
/* update "server A" with the new value.                     */
/* Read a value from "server B". Add 1 to the value and      */
/* update "server B" with the new value.                     */
/*                                                           */
/* Check that all requests were successful. If not, request  */
/* a rollback.                                               */
/*                                                           */
/* End the two-phase commit transaction by calling podtxne.  */
/*                                                           */
/* If all requests were successful, the next GU call will    */
/* trigger the two-phase commit.                             */
/*                                                           */
/*************************************************************/

Process_transaction: PROC;
/* Begin a transaction. */
call podtxnb;
if check_errors('podtxnb') ^= completion_status_yes then
  return;
put skip list('Two-phase commit transaction begins');
call read_value_A;
if read_result_A = good_result then
  do;
    call update_value_A;
  end;
if update_result_A = good_result then
  do;
Example 4: The DATAC Demonstration Module (Sheet 5 of 10)

call read_value_B;
end;

if read_result_B = good_result
then
do;
call update_value_B;
end;

if read_result_A = good_result &
update_result_A = good_result &
read_result_B = good_result &
update_result_B = good_result
then
do;
dc_text_area.dc_text_msg.header =
'Two-phase commit transaction completed';
dc_text_area.dc_text_msg.result = ' ';
dc_text_area.msg_length = 42;
put skip list('All updates successful -');
put skip list('request commit');
end;
else
do;
dc_text_area.dc_text_msg.header =
'A problem was encountered - rolling back';
dc_text_area.dc_text_msg.result = ' ';
dc_text_area.msg_length = 44;
put skip list('Some updates were not successful -');
put skip list('request rollback');
call rollback;
end;

/* End the transaction. */
call podtxne;
if check_errors('podtxne') ^= completion_status_yes then
return;
put skip list('Two-phase commit transaction ends');
call insert;
call GET_UNIQ;
END Process_transaction;
**Example 4:** The DATAC Demonstration Module (Sheet 6 of 10)

```plaintext
**************************************************************
/*                                                           */
/* read_value_A                                              */
/*                                                           */
/* Read a value from "server A".                             */
/*                                                           */
**************************************************************
read_value_A: PROC;
call podexec(DataObject_objA,
             read_operation,
             addr(read_operation_args),
             no_user_exceptions);
if check_errors('podexec') = completion_status_yes then
do;
  read_result_A = 1;
  put skip list('Successfully read a value from Server A: ');
  put list(read_operation_args.idl_value);
end;
END read_value_A;

**************************************************************
/*                                                           */
/* update_value_A                                            */
/*                                                           */
/* Request that "server A" update a value.                   */
/*                                                           */
**************************************************************
update_value_A: PROC;
write_operation_args.idl_value = read_operation_args.idl_value + 1;
put skip list('New value for server A: ');
put list(write_operation_args.idl_value);
call podexec(DataObject_objA,
             write_operation,
             addr(write_operation_args),
             no_user_exceptions);
if check_errors('podexec') = completion_status_yes then
```

---

Example 4: The DATAC Demonstration Module (Sheet 6 of 10)
Example 4: The DATAC Demonstration Module (Sheet 7 of 10)

do;
update_result_A = 1;
put skip list('Server A has successfully updated the value.');
end;
END update_value_A;

/*************************************************************/
/* read_value_B */
/* Read a value from "server B". */
/*************************************************************/
read_value_B: PROC;
call podexec(DataObject_objB,
     read_operation,
     addr(read_operation_args),
     no_user_exceptions);
if check_errors('podexec') = completion_status_yes
then
do;
    read_result_B = 1;
    put skip list('Successfully read a value from Server B: ');
    put list(read_operation_args.idl_value);
end;
END read_value_B;

/*************************************************************/
/* update_value_B */
/* Request that "server B" update a value. */
/*************************************************************/
update_value_B: PROC;
write_operation_args.idl_value = read_operation_args.idl_value + 1;
put skip list('New value for server B: ');
put list(write_operation_args.idl_value);
Example 4: The DATAC Demonstration Module (Sheet 8 of 10)

call podexec(DataObject_objB,
    write_operation,
    addr(write_operation_args),
    no_user_exceptions);

if check_errors('podexec') = completion_status_yes
    then
        do;
            update_result_B = 1;
            put skip list('Server B has successfully updated the value.');
        end;
    END update_value_B;

END;
Example 4: The DATAC Demonstration Module (Sheet 9 of 10)

do;
    dc_text_area.dc_text_msg.header = 'Output from transaction: ';
    dc_text_area.dc_text_msg.result = in_trancode;
    call write_dc_text(string(dc_text_area.dc_text_msg), 49);
end;
END GET_UNIQ;

/***************************************************************************/
/*                                                               */
/* Insert                                                           */
/*                                                               */
/* Issue an INSERT call.                                          */
/*                                                               */
/***************************************************************************/
INSERT: PROC;
    call write_dc_text(string(dc_text_area.dc_text_msg),
                        dc_text_area.msg_length);
END INSERT;

/***************************************************************************/
/*                                                               */
/* Rollback                                                       */
/*                                                               */
/* Issue a ROLLBACK call.                                         */
/*                                                               */
/***************************************************************************/
ROLLBACK: PROC;
    call plitdli(two,rolb,pcblist.io_pcb_ptr);
    if io_pcb.status_code ^= ' ' then
        do;
            put skip list('ROLLBACK FAILED with status code error of ');
            put list(io_pcb.status_code);
        end;
    end;
END ROLLBACK;

/***************************************************************************/
Explanation of the DATAC module

The IMS DATAC module can be explained as follows:

1. data-urlA and data-urlB define corbaloc URL strings in the corbaloc:rir format. These strings identify the servers with which the client is to communicate. The strings can be passed as parameters 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. You can use the check_errors function to check 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 to server 'A' is copied.
6. **STR2OBJ** is called to create an object reference to the server 'A' 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:/DataObjectA` (as defined in point 1). See “**STR2OBJ**” on page 481 for more details of the various forms of corbaloc URL strings and the ways you can use them.
7. **STRSET** is called to create an unbounded string to which the stringified object reference to server 'B' is copied.
8. **STR2OBJ** is called to create an object reference to the server 'B' 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:/DataObjectB` (as defined in point 1). See “**STR2OBJ**” on page 481 for more details of the various forms of corbaloc URL strings and the ways you can use them.
9. **PODtxnB** is called to indicate the start of two-phase commit processing. The next APPC conversation with the client adapter, which is established at the next call to **PODEXEC**, will be at sync level 2.
10. **PODEXEC** is called in this procedure to read a value from server 'A'.
11. **PODEXEC** is called in this procedure to update a value from server 'A'. Server 'A' will log that an update has been requested, but make no actual changes.
12. **PODEXEC** is called in this procedure to read a value from server 'B'.
13. **PODEXEC** is called in this procedure to update a value from server 'B'. Server 'B' will log that an update has been requested, but make no actual changes.
14. If any call to **PODEXEC** was unsuccessful, ask IMS to initiate rollback processing to undo the updates made by the servers. Server 'A' and 'B' will destroy the log that was holding the potential updates. No actual updates will be made.
15. **PODTXNE** is called to indicate the end of two-phase commit processing. This requests that APPC deallocates the conversation. However, the actual deallocation does not occur until the two-phase commit processing has completed.

16. The IMS transaction ends. This triggers the start of two-phase commit processing. The client adapter is notified that the IMS transaction has initiated two-phase commit processing. The client adapter requests that server 'A' and server 'B' prepare their updates. Each server replies to the client adapter that they are either able or unable to commit the update. If either server replies that they are unable to commit the update, each server is asked to roll back and destroy the log that was holding the potential update. If both servers reply that they are able to commit the changes, the client adapter requests each server to commit their changes. The APPC conversation between IMS and the client adapter deallocates, and two-phase commit processing ends.

---

**Location of the DATAC module**

You can find a complete version of the IMS **DATAC** client module in `orbixhlq.DEKOS.IMS.PLI.SRC(DATAC)`. 
### Building the Client

| JCL to run the Orbix IDL compiler | Before you can build the client, you must run the Orbix IDL compiler on the IDL supplied in `orbixhlq.DEMOS.IDL(DATA)`. Sample JCL to do this can be found in `orbixhlq.DEMOS.PLI.IMS.BLD.JCLLIB(DATAIDL)`. |
| JCL to build the client           | Sample JCL used to compile and link the client can be found in `orbixhlq.DEMOS.IMS.PLI.BLD.JCLLIB(DATACB)`. |
| Resulting load module            | When the JCL has successfully executed, it results in a load module that is contained in `orbixhlq.DEMOS.IMS.PLI.LOADLIB(DATACL)`. |
## Building the Servers

### JCL to run the Orbix IDL compiler
Before you can build the servers, ensure that you have run the Orbix IDL compiler on the IDL supplied in `orbixhlq.DEMOS.IDL(DATA)`. Sample JCL to do this can be found in `orbixhlq.DEMOS.IMS.PLI.BLD.JCLLIB(DATAIDL)`.

**Note:** If you have already built the client, this step should have already been completed.

### JCL to build the servers
Sample JCL used to compile and link the servers can be found in `orbixhlq.DEMOS.CPP.BLD.JCLLIB(DATASV)`.

### Resulting load module
When the JCL has successfully executed, it results in a load module that is contained in `orbixhlq.DEMOS.CPP.LOADLIB(DATASV)`.
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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Define a transaction to IMS.</td>
</tr>
<tr>
<td>2</td>
<td>Provide the IMS client load module to the IMS region.</td>
</tr>
<tr>
<td>3</td>
<td>Start the locator, node daemon, and RRS OTSTM on the server host.</td>
</tr>
<tr>
<td>4</td>
<td>Start the batch servers.</td>
</tr>
<tr>
<td>5</td>
<td>Customize the batch server IORs.</td>
</tr>
<tr>
<td>6</td>
<td>Configure and run the client adapter.</td>
</tr>
</tbody>
</table>

Step 1—Define a transaction to 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:

```
APPLCN GPSB=DATAACL,
         PGMTY=(TP,,2),
         SCHDTYP=PARALLEL
         LANG=PLI
         TRANSACT CODE=DATAACL,
         EDIT=(ULC)
```

Step 2—Provide client load module to IMS region
Ensure that the `orbixhlg.DEMOS.IMS.PLI.LOADLIB` PDS is added to the STEPLIB for the IMS region that is to run the transaction.
Step 3—Start locator, node daemon, and RRS OTSTM on server

This step assumes that you intend running the IMS client against the demonstration batch 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.JCLLIB(LOCATOR)`.
2. Start the node daemon by submitting `orbixhlq.JCLLIB(NODEDAEM)`.
3. Start the RRS OTSTM server by submitting `orbixhlq.JCLLIB(OTSTM)`.

See “Running the Server and Client” on page 47 for more details of running the locator and node daemon on the batch server host.

See the chapter on Using OTS RRS Transaction Manager in the Mainframe OTS Guide for more details of running the RRS OTSTM server.

Step 4—Start batch servers

This step assumes that you intend running the IMS client against the demonstration batch servers.

Submit the `orbixhlq.DEVOS.CPP.RUN.JCLLIB(DATAA)` and `orbixhlq.DEVOS.CPP.RUN.JCLLIB(DATAB)` JCL to start the batch servers.

Step 5—Customize batch server IORs

When you run the demonstration batch servers they publish their IORs to `orbixhlq.DEVOS.IORS(DATAA)` and `orbixhlq.DEVOS.IORS(DATAB)`.

The demonstration IMS client needs to use these IORs to contact the demonstration batch servers. The demonstration IMS client obtains the object reference for the demonstration batch servers in the form of a corbaloc URL string. A corbaloc URL string can take different formats. For the purposes of this demonstration, the corbalocs take the form `corbaloc:rir:/DataObjectA` and `corbaloc:rir:/DataObjectB`.

This form of the corbaloc URL string requires the use of the configuration variables, `initial_references:DataObjectA:reference` and `initial_references:DataObjectB:reference`, in the configuration domain.

When you submit the JCL in `orbixhlq.DEVOS.IMS.PLL.BLD.JCLLIB (DATATORS)`, it automatically adds these configuration entries to the configuration domain:

```plaintext
initial_references:DataObjectA:reference = "IOR...";
initial_references:DataObjectB:reference = "IOR...";
```
The IOR values are taken from `orbixhlq.DEMOS.IORS(DATAA)` and `orbixhlq.DEMOS.IORS(DATAB)`.

See “STR2OBJ” on page 481 for more details of the various forms of corbaloc URL strings and the ways you can use them.

### Step 6—Configure and run client adapter

The client adapter must now be configured before you can start the client (the 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 `orbixhlq.JCLLIB(IMSCA)`.

**Note:** See “Running an IMS Two-Phase Commit Client against Batch Servers” on page 117 for details of how to run the sample two-phase commit client demonstration.
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:

<table>
<thead>
<tr>
<th>Demonstration</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running a Batch Client against an IMS Server</td>
<td>115</td>
</tr>
<tr>
<td>Running an IMS Client against a Batch Server</td>
<td>116</td>
</tr>
<tr>
<td>Running an IMS Two-Phase Commit Client against Batch Servers</td>
<td>117</td>
</tr>
</tbody>
</table>
Running a Batch Client against an 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 79 have been successfully completed.
2. Run the batch client as described in “Running the Server and Client” on page 45.

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 an IMS Client against a 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 89 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
```
Running an IMS Two-Phase Commit Client against Batch Servers

Overview
This subsection describes what you need to do to successfully run the demonstration IMS two-phase commit client against the demonstration batch servers. It also provides an overview of the output produced.

Note: For instructions on recovery processing for any unsuccessful runs of an application, see `orbiNLQ.Demos.ims.pll.readme(DATACL)`.

Steps
The steps to run the demonstration IMS two-phase commit client against the demonstration batch servers are:

1. Ensure that all the steps in “Preparing the Client to Run in IMS” on page 111 have been successfully completed.
2. Run the IMS client by entering the transaction name, DATACL, in the relevant IMS region.

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

```
Initializing the ORB
Registering the Interface
Two-phase commit transaction begins
Invoking: read:IDL:Data:1.0
Successfully read a value from server A: 0000000001
New value for server A: 0000000002
Invoking: write:IDL:Data:1.0
Server A has successfully updated the value.
Invoking: read:IDL:Data:1.0
Successfully read a value from server B: 0000000001
New value for server B: 0000000002
Invoking: write:IDL:Data:1.0
Server B has successfully updated the value.
All updates are successful - request commit
Two-phase commit transaction ends
```
The IMS client sends the following output to the IMS message queue:

<table>
<thead>
<tr>
<th>Output from transaction: DATACL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two-phase commit transaction completed</td>
</tr>
</tbody>
</table>

**Batch server 'A' output**

OTS Recovery Demo Server
Initializing the ORB
Server ID is A
IOR file is DD:IORS(DATAA)
Data file is DD:DATA(DATAA)
Log file is DD:DATA(LOGA)
Resolving Transaction
Resolving RootPOA
Creating POA with REQUIRES OTS Policy
Creating POA with lifespan policy of PERSISTENT
Creating POA with an ID assignment of USER
Creating Data servant and object
Creating POA for Resource objects
Reading data from file DD:DATA(DATAA)
Value is 1
Writing object reference to DD:IORS(DATAA)
Activation POA for Data object
Data servant read() called
Read-only access: not registering Resource object
Current value is 1
Data servant write() called
Getting coordinator for current transaction
Getting Transaction Identifier
Creating Resource servant
Activating Resource object
Registering Resource object with coordinator
Activating the Resource POA
Setting value to 2
Resource servant prepare() called
Voting to commit the transaction
Writing prepare record
Resource servant commit() called
Writing data to file DD:DATA(DATAA)
Deleting prepare record
Deactivating Resource object
Resource servant destructed
Batch server 'B' output

Batch server 'B' produces the following output:

OTS Recovery Demo Server
Initializing the ORB
Server ID is B
IOR file is DD:IORS(DATAB)
Data file is DD:DATA(DATAB)
Log file is DD:DATA(LOGB)
Resolving TransactionCurrent
Resolving RootPOA
Creating POA with REQUIRES OTS Policy
Creating POA with lifespan policy of PERSISTENT
Creating POA with an ID assignment of USER
Creating Data servant and object
Creating POA for Resource objects
Reading data from file DD:DATA(DATAB)
Value is 1
Writing object reference to DD:IORS(DATAB)
Activation POA for Data object
Data servant read() called
Read-only access: not registering Resoure object
Current value is 1
Data servant write() called
Getting coordinator for current transaction
Getting Transaction Identifier
Creating Resource servant
Activating Resource object
Registering Resource object with coordinator
Activating the Resource POA
Setting value to 2
Resource servant prepare() called
Voting to commit the transaction
Writing prepare record
Resource servant commit() called
Writing data to file DD:DATA(DATAB)
Deactivating Resource object
Deactivating Resource servant destructed
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. Additionally, this chapter shows how to develop a CICS client that supports two-phase commit transactions.

In this chapter

This chapter discusses the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>123</td>
</tr>
<tr>
<td>Developing the Application Interfaces</td>
<td>131</td>
</tr>
<tr>
<td>Developing the CICS Server</td>
<td>143</td>
</tr>
<tr>
<td>Developing the CICS Client</td>
<td>155</td>
</tr>
<tr>
<td>Developing the CICS Two-Phase Commit Client</td>
<td>166</td>
</tr>
<tr>
<td>Running the Demonstrations</td>
<td>186</td>
</tr>
</tbody>
</table>
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 the following Orbix PL/I applications:

- CICS server
- CICS client
- CICS two-phase commit client

It also introduces the following PL/I demonstrations that are supplied with your Orbix Mainframe installation, and outlines where you can find the various source code and JCL elements for them:

- SIMPLE CICS server
- SIMPLE CICS client
- DATAC CICS two-phase commit client

Steps to create an application

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

1. “Developing the Application Interfaces” on page 131.
2. “Developing the CICS Server” on page 143.
3. “Developing the CICS Client” on page 155.
4. “Developing the CICS Two-Phase Commit Client” on page 166.

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. Additionally, this chapter describes how to develop an Orbix PL/I two-phase commit CICS client, and run it against two C++ servers. The supplied demonstrations do not reflect real-world scenarios requiring 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 object. 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.

### The demonstration CICS two-phase commit client

The Orbix PL/I two-phase commit client developed in this chapter runs in a CICS region. It uses the clearly defined object interface, Data, to access and update data from two Orbix C++ batch servers. When the client invokes a remote operation, a request message is sent from the client to one of the servers 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 servers.

### 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 14 provides a summary of these code elements and JCL components (where orbixhlq represents your installation’s high-level qualifier).

**Table 14: Supplied Code and JCL (Sheet 1 of 4)**

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orbixhlq.DEMOS.IDL(SIMPLE)</td>
<td>This is the supplied IDL for the simple CICS client and server.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.IDL(DATA)</td>
<td>This is the supplied IDL for the CICS two-phase commit client.</td>
</tr>
</tbody>
</table>
Overview

Table 14: Supplied Code and JCL (Sheet 2 of 4)

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>orbixhlq.DEMOS.CICS.PLI.SRC</code> (SIMPLEV)</td>
<td>This is the source code for the CICS server mainline module, which is generated when you run the JCL in <code>orbixhlq.DEMOS.CICS.PLI.BLD.JCLLIB(SIMPLIDL)</code>. (The CICS server mainline code is not shipped with the product. You must run the SIMPLIDL JCL to generate it.)</td>
</tr>
<tr>
<td><code>orbixhlq.DEMOS.CICS.PLI.SRC</code> (SIMPLEI)</td>
<td>This is the source code for the CICS server implementation module.</td>
</tr>
<tr>
<td><code>orbixhlq.DEMOS.CICS.PLI.SRC</code> (SIMPLEC)</td>
<td>This is the source code for the CICS client module.</td>
</tr>
<tr>
<td><code>orbixhlq.DEMOS.CICS.PLI.SRC</code> (DATAC)</td>
<td>This is the source code for the CICS two-phase commit client module.</td>
</tr>
<tr>
<td><code>orbixhlq.DEMOS.CICS.PLI.BLD.JCLLIB(SIMPLIDL)</code></td>
<td>This JCL runs the Orbix IDL compiler. See “Orbix IDL Compiler” on page 134 for more details of this JCL and how to use it.</td>
</tr>
<tr>
<td><code>orbixhlq.DEMOS.CICS.PLI.BLD.JCLLIB</code> (DATAIDL)</td>
<td>This JCL runs the Orbix IDL compiler for the CICS two-phase commit client.</td>
</tr>
<tr>
<td><code>orbixhlq.DEMOS.CICS.PLI.BLD.JCLLIB</code> (SIMPLECB)</td>
<td>This JCL compiles the CICS simple client module to create the SIMPLE client program.</td>
</tr>
<tr>
<td><code>orbixhlq.DEMOS.CICS.PLI.BLD.JCLLIB</code> (DATACB)</td>
<td>This JCL compiles the CICS two-phase commit client module.</td>
</tr>
<tr>
<td><code>orbixhlq.DEMOS.CICS.PLI.BLD.JCLLIB</code> (SIMPLIOR)</td>
<td>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.</td>
</tr>
</tbody>
</table>
This JCL adds the following configuration entry to the configuration member:

```java
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.DEVOS.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

```java
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 481 for more details of the various forms of corbaloc URL strings and the ways you can use them.

This JCL adds the following configuration entries to the configuration member:

```java
initial_references:DataObjectA:reference="IOR...";
initial_references:DataObjectB:reference="IOR...";
```

These configuration entries specify the IORs that the CICS two-phase commit client uses to contact the C++ batch servers. The IORs that are set as the value for these configuration entries are the IORs that are published in `orbixhlq.DEVOS.IORS(DATAA)` and `orbixhlq.DEVOS.IORS(DATAB)` when you run the C++ batch servers.

The object references for the servers are represented to the demonstration CICS two-phase commit client as corbaloc URL strings in the form `corbaloc:rir:/DataObjectA` and `corbaloc:rir:/DataObjectB`. This form of corbaloc URL string requires the use of the `initial_references:DataObjectA:reference="IOR..."` and `initial_references:DataObjectB:reference="IOR..."` configuration items.
<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orbixhlg.JCLLIB(CICSCA)</td>
<td>This JCL runs the CICS client adapter.</td>
</tr>
<tr>
<td>orbixhlg.JCLLIB(CICSA)</td>
<td>This JCL runs the CICS server adapter.</td>
</tr>
<tr>
<td>orbixhlg.DEMOS.CPP.BLD.JCLLIB (DATASV)</td>
<td>This JCL builds the C++ servers for the CICS two-phase commit client.</td>
</tr>
<tr>
<td>orbixhlg.DEMOS.CPP.BLD.JCLLIB (DATAA)</td>
<td>This JCL runs the C++ server 'A' for the CICS two-phase commit client.</td>
</tr>
<tr>
<td>orbixhlg.DEMOS.CPP.BLD.JCLLIB (DATAB)</td>
<td>This JCL runs the C++ server 'B' for the CICS two-phase commit client.</td>
</tr>
<tr>
<td>orbixhlg.DEMOS.CPP.GEN</td>
<td>This PDS contains generated stub code for the C++ servers.</td>
</tr>
<tr>
<td>orbixhlg.DEMOS.CPP.H</td>
<td>This PDS contains C++ header files.</td>
</tr>
<tr>
<td>orbixhlg.DEMOS.CPP.HH</td>
<td>This PDS contains IDL generated header files.</td>
</tr>
<tr>
<td>orbixhlg.DEMOS.CPP.LOADLIB</td>
<td>This PDS contains the C++ server module for the two-phase commit CICS client.</td>
</tr>
<tr>
<td>orbixhlg.DEMOS.CPP.SRC</td>
<td>This PDS contains the C++ server module source code for the two-phase commit CICS client.</td>
</tr>
<tr>
<td>orbixhlg.DEMOS.CPP.TWOPCA</td>
<td>This PDS contains the data store for the two-phase commit C++ server 'A'.</td>
</tr>
<tr>
<td>orbixhlg.DEMOS.CPP.TWOPCB</td>
<td>This PDS contains the data store for the two-phase commit C++ server 'B'.</td>
</tr>
</tbody>
</table>
Supplied include members

Table 15 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>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>orbixhlq</em>.INCLUDE.PLINCL(CHKCLC)</td>
<td>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.</td>
</tr>
<tr>
<td><em>orbixhlq</em>.INCLUDE.PLINCL(CHKCICS)</td>
<td>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.</td>
</tr>
<tr>
<td><em>orbixhlq</em>.INCLUDE.PLINCL(CHKERRS)</td>
<td>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.</td>
</tr>
<tr>
<td><em>orbixhlq</em>.INCLUDE.PLINCL(CORBA)</td>
<td>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 <code>&amp;client_only='yes'</code>.</td>
</tr>
<tr>
<td><em>orbixhlq</em>.INCLUDE.PLINCL(CORBACOM)</td>
<td>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.</td>
</tr>
<tr>
<td><em>orbixhlq</em>.INCLUDE.PLINCL(CORBASV)</td>
<td>This is relevant to CICS servers. It contains PL/I runtime function definitions that can be used by servers.</td>
</tr>
</tbody>
</table>
| *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*.
### Table 15: Supplied Include Members (Sheet 2 of 2)

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orbixhlq.INCLUDE.PLINCL (EXCNAME)</td>
<td>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.</td>
</tr>
<tr>
<td>orbixhlq.INCLUDE.PLINCL (URLSTR)</td>
<td>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 481 for more details.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.CICS.PLI.PLINCL</td>
<td>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.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.CICS.MFAMAP</td>
<td>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 qualified 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.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.TYPEINFO</td>
<td>This PDS is relevant to CICS servers only. It is empty at installation time. It is used to store the type information that is generated when you run the JCL to run the Orbix IDL compiler for the supplied demonstrations. The contents of the type information member describe the contents of the given IDL file from which it was generated.</td>
</tr>
</tbody>
</table>
Checking JCL components

When creating the CICS simple client or server, or the CICS two-phase commit client, 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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
</table>
| 1    | Define public IDL interfaces to the objects required in your system.  
      | See “Defining IDL Interfaces” on page 132. |
| 2    | Use the ORXCOPY utility to copy your IDL files to z/OS (if necessary).  
      | See “ORXCOPY Utility” on page 523. |
| 3    | Run the Orbix IDL compiler to generate PL/I include members, server source, and server mapping member.  
      | See “Orbix IDL Compiler” on page 134. |
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.DEMS.IDL(SIMPLE):

```idl
// 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.DEMS.IORS(SIMPLE). In this case, the CICS server adapter IOR is published to orbixhlq.DEMS.IORS(SIMPLE) when you run `orbixhlq.DEMS.CICS.PL/I.BLD.JCLLIB(SIMPLIOR)`.

The CICS demonstration client of the batch demonstration server locates the SimpleObject object by reading the IOR for the batch server from orbixhlq.DEMS.IORS(SIMPLE). In this case, the batch server IOR is
published to orbixhq.DEMOS.ORS(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:** If your IDL files are not already contained in z/OS data sets, you must copy them to z/OS before you proceed. You can use the ORXCOPY utility to do this. If necessary, see “ORXCOPY Utility” on page 523 for more details.

**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 293 for more details.

Example of the SIMPLIDL JCL

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

```plaintext
//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.PROCLIB)
//   INCLUDE MEMBER=(ORXVARS)
//*
//IDLPLI EXEC ORXIDL,
// SOURCE=SIMPLE,
// IDL=ORBIX..DEMO2.IDL,
```
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:-inf'` 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:-ttran_or_program_name` arguments.
  - Type information for the SIMPLE IDL member via the `-inf` sub-argument to the `-mfa` argument.

  **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:-inf'` is relevant to CICS server development for APPC. This line generates the same items as the `IDLPARM='-pli:-TCICS -mfa:-tSIMPLESV:-inf'`. 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:-inf' 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, a CICS server adapter mapping member for EXCI, and type
information for the `SIMPLE` IDL member.

See “Orbix IDL Compiler” on page 293 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:

```
orbi$hlg.DEVOS.CICS.PLI.BLD.JCLLIB(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 235 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.
Table 16 shows the PL/I include members that the Orbix IDL compiler generates, based on the defined IDL.

### Table 16: Generated PL/I Include Members (Sheet 1 of 2)

<table>
<thead>
<tr>
<th>Copybook</th>
<th>JCL Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idlmembernameD</td>
<td>COPYLIB</td>
<td>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.</td>
</tr>
<tr>
<td>idlmembernameE</td>
<td>COPYLIB</td>
<td>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 idlmembernameD include member.</td>
</tr>
<tr>
<td>idlmembernameH</td>
<td>COPYLIB</td>
<td>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 idlmembernameE include member. This member is automatically included in the idlmembernameE include member.</td>
</tr>
</tbody>
</table>
### Table 16: Generated PL/I Include Members (Sheet 2 of 2)

<table>
<thead>
<tr>
<th>Copybook</th>
<th>JCL Keyword Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idlmembernameT</td>
<td>COPYLIB</td>
<td>This include member contains the based structure declarations that are used in the <code>idlmembernameT</code> include member. This member is automatically included in the <code>idlmembernameM</code> include member.</td>
</tr>
<tr>
<td>idlmembernameX</td>
<td>COPYLIB</td>
<td>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 <code>idlmembernameV</code> source code member.</td>
</tr>
<tr>
<td>idlmembernameD</td>
<td>COPYLIB</td>
<td>This include member contains a select statement for calling the correct procedure for the requested operation. This include member is automatically included in the <code>idlmembernameI</code> source code member.</td>
</tr>
</tbody>
</table>
## Generated server source members

Table 17 shows the server source code members that the Orbix IDL compiler generates, based on the defined IDL:

### Table 17: Generated Server Source Code Members

<table>
<thead>
<tr>
<th>Member</th>
<th>JCL Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idlmembernameI</td>
<td>IMPL</td>
<td>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 <code>-S</code> and <code>-TCICS</code> arguments with the IDL compiler.</td>
</tr>
<tr>
<td>idlmembernameV</td>
<td>IMPL</td>
<td>This is the CICS server mainline source code member. It is generated by default. However, you can use the <code>-V</code> argument with the IDL compiler, to prevent generation of this member.</td>
</tr>
</tbody>
</table>

**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 293 for more details of the IDL compiler arguments used to generate, and prevent generation of, CICS server source code.
Developing the Application Interfaces

Generated server adapter mapping member

Table 18 shows the CICS server adapter mapping member that the Orbix IDL compiler generates, based on the defined IDL.

Table 18: Generated CICS Server Adapter Mapping Member

<table>
<thead>
<tr>
<th>Copybook</th>
<th>JCL Keyword Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idlmembernameA</td>
<td>IDLMFA</td>
<td>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.</td>
</tr>
</tbody>
</table>

Generated type information member

Table 19 shows the type information member that the Orbix IDL compiler generates, based on the defined IDL.

Table 19: Generated CICS Server Adapter Mapping Member

<table>
<thead>
<tr>
<th>Copybook</th>
<th>JCL Keyword Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idlmembernameB</td>
<td>IDLTYPEI</td>
<td>Type information describing the operation signatures of the interface whose IDL it was generated from.</td>
</tr>
</tbody>
</table>

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.DEOMS.CICS.PLI.PLINCL(SIMPLED)`
- `orbixhlq.DEOMS.CICS.PLI.PLINCL(SIMPLEL)`
- `orbixhlq.DEOMS.CICS.PLI.PLINCL(SIMPLEM)`
- `orbixhlq.DEOMS.CICS.PLI.PLINCL(SIMPLET)`
- `orbixhlq.DEOMS.CICS.PLI.PLINCL(SIMPLEX)`
- `orbixhlq.DEOMS.CICS.PLI.SRC(SIMPLEV)`
- `orbixhlq.DEOMS.CICS.PLI.SRC(SIMPLEI)`

• `orbixhlq.DEMOS.CICS.MFAMAP(SIMPLEA)`
• `orbixhlq.DEMOS.TYPEINFO(SIMPLEB)`

**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.JCLLIB(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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“Writing the Server Implementation” on page 144.</td>
</tr>
<tr>
<td>2</td>
<td>“Writing the Server Mainline” on page 147.</td>
</tr>
<tr>
<td>3</td>
<td>“Building the Server” on page 150.</td>
</tr>
<tr>
<td>4</td>
<td>“Preparing the Server to Run in CICS” on page 151.</td>
</tr>
</tbody>
</table>
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` 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 5: The SIMPLEI Demonstration Module (Sheet 1 of 2)**

```pli
SIMPLEI: PROC;
/*The following line enables the runtime to call this procedure*/
DISPTCH: ENTRY;
dcl (addr,low,sysnull) builtin;
#include CORBA;
#include CHKERRS;
#include SIMPLEM;
#include DISPINIT;

/* ================ Start of global user code =============== */
/* ================= End of global user code ================ */
/* ----------------------------------------------------------*/
/* Dispatcher : select(operation)                            */
/* ----------------------------------------------------------*/
#include SIMPLED;
```
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 6: 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 SETUPSV;
@include CORBA;
@include CHKERRS;
@include SIMPLET;
@include SIMPLEX;

alloc pod_status_information set(pod_status_ptr);
**Example 6: The SIMPLEV Demonstration Module** (Sheet 2 of 2)

```plaintext
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 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.JCLLIB(SIMPLIDL)` to run the Orbix IDL compiler.
# Building the Server

<table>
<thead>
<tr>
<th>Location of the JCL</th>
<th>Sample JCL used to compile and link the CICS server mainline and server implementation is in <code>orbixhlq.DEFOS.CICS.PLI.BLD.JCLLIB(SIMPLES).</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Resulting load module</td>
<td>When this JCL has successfully executed, it results in a load module that is contained in <code>orbixhlq.DEFOS.CICS.PLI.LOADLIB(SIMPLES).</code></td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Define an APPC transaction definition or EXCI program definition for CICS.</td>
</tr>
<tr>
<td>2</td>
<td>Provide the CICS server load module to a CICS region.</td>
</tr>
<tr>
<td>3</td>
<td>Generate mapping member entries for the CICS server adapter.</td>
</tr>
<tr>
<td>4</td>
<td>Add the interface’s operation signatures to the type information repository, stored in the TYPEINFO PDS.</td>
</tr>
<tr>
<td>5</td>
<td>Obtain the IOR for use by the client program.</td>
</tr>
</tbody>
</table>

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:

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

See the supplied `orbixhlq.JCLLIB(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.LOADLIB` 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/Object,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/Object,call_me,SMSV)
```

The generation of a mapping member for the CICS server adapter is performed by the `orbixhlq.DEMOS.CICS.PLI.BLD.JCLLIB(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 operation signatures to type_info store

The CICS server adapter needs to be able to obtain operation signatures for the PL/I server. For the purposes of this demonstration, the TYPEINFO PDS is used to store 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. This information is generated by supplying the -mfa:-inf option to the Orbix IDL compiler, for example, as used in the SIMPLIDL JCL that is used to generate the source and include members for this demonstration.

Note: An IDL interface only needs to be added to the type information store once.

Note: An alternative to using type information files is to use the Interface Repository (IFR). This is an alternative method of allowing the CICS server adapter to retrieve IDL type information. If you are using the IFR, you must ensure that the relevant IDL for the server has been added to the IFR (that is, registered with it) before the CICS server adapter is started.

To add IDL to the IFR, first ensure the IFR is running. You can use the JCL in orbixhlg.JCL(IFR) to start it. Then, in the JCL that you use to run the Orbix IDL compiler, add the line // IDLPARM=-R to register the IDL. In this case, ensure that all other // IDLPARM lines are commented out as follows: /* IDLPARM...”

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 type_info store contains the relevant operation signatures (or, if using the IFR, the IFR server is running and contains the relevant IDL). See “Step 4—Adding operation signatures to type_info store” on page 153 for details of how to populate the type_info store.
- The CICS server adapter mapping member contains the relevant mapping entries. For the purposes of this example, ensure that the orbixhlg.DEMOS.CICS.MFAMAP(SIMPLEA) mapping member is being used. See the CICS Adapters Administrator's Guide for details about CICS server adapter mapping members.
The CICS server adapter is running. See the CICS Adapters Administrator’s Guide for more details of how to start the CICS server adapter, using the supplied JCL in orbix.hlq.JCLLIB(CICSA).

Now submit orbix.hlq.Demos.CICS.PLI.BLD.JCLLIB(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:

```plaintext
// JCLLIB ORDER=(orbix.hlq.PROCLIB)
// 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..DEMONS.IORS(SIMPLE),DISP=SHR
// ORBARGS DD *
// -ORBname iona_utilities.cicsa
// ITDOMAIN DD DSN=&ORBIXCFG(&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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“Writing the Client” on page 156.</td>
</tr>
<tr>
<td>2</td>
<td>“Building the Client” on page 161.</td>
</tr>
<tr>
<td>3</td>
<td>“Preparing the Client to Run in CICS” on page 162.</td>
</tr>
</tbody>
</table>
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 7: The SIMPLEC Demonstration Module (Sheet 1 of 3)

```plaintext
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);

call podstat(pod_status_ptr);

2 /* Initialize our ORB */
call orbargs(arg_list, 
            arg_list_len, 
            orb_name, 
            orb_name_len);
```
Example 7: The SIMPLEC Demonstration Module (Sheet 2 of 3)

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

/* Register the SimpleObject intf with the PL/I runtime */
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 */
call strset(simple_url_ptr,
  simple_url,
  length(simple_url));
if check_errors('strset') ^= completion_status_yes then
  exec cics return;

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...');
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';

EXEC CICS SEND TEXT FROM (MessageText) LENGTH(79) FREEKB;
```
if check_errors('orbargs') ^= completion_status_yes then
  exec cics return;
/* Register the SimpleObject intf with the PL/I runtime */
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 */
call strset(simple_url_ptr,
  simple_url,
  length(simple_url));
if check_errors('strset') ^= completion_status_yes then
  exec cics return;
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...'));
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';
EXEC CICS SEND TEXT FROM (MessageText) LENGTH(79) FREEKB;
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 the `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.

Example 7: The SIMPLEC Demonstration Module (Sheet 3 of 3)

```c++
/* Free the simple_persistent object reference */
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;
```
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 481 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.JCLLIB(SIMPLECB)`.

**Resulting load module**
When the JCL has successfully executed, it results in a load module that is contained in `orbixhlq.DEMOS.CICS.PLI.LOADLIB(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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Define an APPC transaction definition for CICS.</td>
</tr>
<tr>
<td>2</td>
<td>Provide the CICS client load module to a CICS region.</td>
</tr>
<tr>
<td>3</td>
<td>Start the locator and node daemon on the server host.</td>
</tr>
<tr>
<td>4</td>
<td>Add the interface’s operation signatures to the type information repository.</td>
</tr>
<tr>
<td>5</td>
<td>Start the batch server.</td>
</tr>
<tr>
<td>6</td>
<td>Customize the batch server IOR.</td>
</tr>
<tr>
<td>7</td>
<td>Configure and run the client adapter.</td>
</tr>
</tbody>
</table>

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)
```
Developing the CICS Client

See the supplied orbixhlq.JCLLIB(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.LoadLib 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 and node daemon on server host**

This step assumes 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.JCLLIB(LOCATOR).
2. Start the node daemon by submitting orbixhlq.JCLLIB(NODEDAEM).

See “Running the Server and Client” on page 45 for more details of running the locator and node daemon on the batch server host.

**Step 4—Add operation signatures to type_info store**

The client adapter needs to be able to know what data types it can expect to marshal from the IMS transaction, and what data types it should expect back from the batch server. This can be done by creating a type information file by running the IDL compiler with the -mfa:-inf flag, which is included in orbixhlq.Demos.CICS.PLI.BLD.JCLLIB(SIMPLIDL). The type information file contains descriptions of the interface's operation signatures (that is, information about the type and direction of the operation parameters, the number of parameters, and whether or not an operation has a return type).

Before the client adapter is run, the TYPEINFO DD card needs to be updated to the location of the TYPEINFO PDS (for the purposes of this example, it should be updated to orbixhlq.Demos.TYPEINFO).

**Note:** An IDL interface only needs to be added to the type information store once.
Step 5—Start batch server

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

Submit the following JCL to start the batch server:

```
orbixhlq.DEOMS.PLI.RUN.JCLLIB(SIMPLESV)
```

See “Running the Server and Client” on page 45 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.DEOMS.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.DEOMS.CICS.PLI.BLD.JCLLIB(UPDTCONF)`, it automatically adds this configuration entry to the configuration domain:

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

The IOR value is taken from the `orbixhlq.DEOMS.IORS(SIMPLE)` member.

Note: An alternative to using type information files is to use the Interface Repository (IFR). This is an alternative method of allowing the client adapter to obtain information about relevant data types. If you are using the IFR, you must ensure that the relevant IDL for the server has been added to the IFR (that is, registered with it) before the client adapter is started.

To add IDL to the IFR, first ensure the IFR is running. You can use the JCL in `orbixhlq.JCL(IFR)` to start it. Then, in the JCL that you use to run the Orbix IDL compiler, add the line `// IDLPARM='R'` to register the IDL. In this case, ensure that all other `// IDLPARM` lines are commented out as follows: `/** IDLPARM...`
Developing the CICS Client

See “STR2OBJ” on page 481 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 (the 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 orbix.hlq.JCLLIB(CICSCA).

**Note:** See “Running the Demonstrations” on page 186 for details of how to run the sample demonstration.
Developing the CICS Two-Phase Commit Client

Overview

This section describes the steps you must follow to develop the CICS two-phase commit client executable for your application. The CICS two-phase commit client developed in this example will connect to two demonstration C++ batch servers.

Steps to develop the client

The steps to develop and run the client application are:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“Writing the Client” on page 167.</td>
</tr>
<tr>
<td>2</td>
<td>“Building the Client” on page 181.</td>
</tr>
<tr>
<td>3</td>
<td>“Building the Servers” on page 182.</td>
</tr>
<tr>
<td>4</td>
<td>“Preparing the Client to Run in CICS” on page 183.</td>
</tr>
</tbody>
</table>
Writing the Client

The client program
The next step is to write the CICS client transaction. This example uses the supplied DATAC client demonstration.

CICS transaction design
A CICS transaction that uses two-phase commit can be broken down as follows:
• Operations that do not require two-phase commit.
• Operations that require two-phase commit.

Read-only operations to local databases or remote servers do not require two-phase commit processing. These operations should be performed first in the CICS transaction ahead of the two-phase commit operations. The rationale behind this is that if operations not requiring two-phase commit processing fail, it might be pointless to perform operations that do require two-phase commit processing.

Overview of CICS transaction layout
Figure 5 provides an overview of CICS transaction layout.

Figure 5: Overview of CICS Transaction Layout
Designing a CICS two-phase commit transaction

When designing a CICS two-phase commit transaction, structure the transaction as follows:

1. Begin the CICS transaction by performing standard Orbix Mainframe CICS client initialization.
2. Perform operations that do not require two-phase commit. If any of the operations fail, skip the two-phase commit processing.
3. Call `PODTXNB` to indicate the start of two-phase commit processing.
4. Call `PODEXEC` (perhaps multiple times) to send an update to a remote server. If any of the calls fail, call rollback and skip any updates to local resources.
5. Make updates to local resources, such as updating a local database. If any of the local updates fail, call rollback.
6. Call `PODXNE` to indicate the end of the two-phase commit work.
7. Call `SYNCPD` to initiate two-phase commit processing.
8. Perform any post two-phase commit work, such as sending a message back to the user.

Commit or rollback scenarios

When a CICS transaction makes updates to resources (that is, local databases or remote CORBA servers) via the client adapter, the updates are not made permanent until the two-phase commit has been successfully processed. The trigger for starting the two-phase commit is when the CICS transaction calls `SYNCPD`.

The client adapter sends a "prepare" message to each remote server that has been updated from the CICS transaction. Each server returns a vote to the client adapter. A vote of "commit" indicates the remote server is willing to commit its updates. A vote of "rollback" indicates the remote server has a problem and that it wants to roll back the update.

The various scenarios that might arise are as follows:

- **Successful two-phase commit**
  
  If all returned votes are "commit", the client adapter calls the IBM API `SRRCMIT`, to inform CICS that all remote servers are willing to commit their updates. If the return code from `SRRCMIT` is 0, the client adapter sends a "commit" message to each remote server. Two-phase commit processing is then completed and all resources are updated.
Developing the CICS Two-Phase Commit Client

• Rollback two-phase commit—Scenario 1
  If the client adapter receives at least one returned vote of "rollback", all updates should be rolled back. The client adapter calls the IBM API \texttt{SRRBACK}, to inform CICS that there are problems. This causes the \texttt{SYNCPOINT} call issued in the CICS transaction to complete with a \texttt{ROLLEDBACK} code.

• Rollback two-phase commit—Scenario 2
  If all returned votes are "commit", the client adapter calls the IBM API \texttt{SRRCMIT}, to inform CICS that all remote servers are willing to commit their updates. If the return code from \texttt{SRRCMIT} is not 0, the client adapter sends a "rollback" message to each server. In this case, this means that a resource other than the remote servers has voted "rollback".

• Rollback two-phase commit—Scenario 3
  If the CICS transaction makes an update to a remote server, and the update fails (because, for example, the server is not running), the transaction calls "rollback" to undo any updates. The client adapter receives the rollback signal and sends a "rollback" message to each server.

Example of the DATACL module

The following is an example of the CICS \texttt{DATAC} module:

\textbf{Example 8: The DATAC Demonstration Module} (Sheet 1 of 9)

\begin{verbatim}
DATAC: PROC OPTIONS (MAIN NOEXECOPS);

%client_only='yes';

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

dcl arg_list char(40) init('');

dcl orb_name char(9) init('twopc_orb');

dcl data_urlA char(26) init('corbaloc:rir:/DataObjectA ');

DATAC: PROC OPTIONS (MAIN NOEXECOPS);

%client_only='yes';

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

dcl arg_list char(40) init('');

dcl orb_name char(9) init('twopc_orb');

dcl data_urlA char(26) init('corbaloc:rir:/DataObjectA ');
\end{verbatim}
Example 8: The DATAC Demonstration Module (Sheet 2 of 9)

dcl data_urlB                  char(26)
  init('corbaloc:rir:/DataObjectB ');
dcl data_url_ptr               ptr  init(sysnull());
dcl DataObject_objA            ptr;
dcl DataObject_objB            ptr;
dcl read_result_A              fixed bin(31)  init(0);
dcl update_result_A            fixed bin(31)  init(0);
dcl read_result_B              fixed bin(31)  init(0);
dcl update_result_B            fixed bin(31)  init(0);
dcl good_result                fixed bin(31)  init(1);
dcl MessageText                char(79) init('');
%include CORBA;
%include CHKCLCIC;
%include DATAM;
%include DATAX;
/****************************/
/*                         */
/* Process, a two-phase commit transaction. The general flow */
/* of the transaction is as follows:                        */
/* begin a transaction (PODTXNB)                            */
/* read a value from "server A" (PODEXEC)                   */
/* send an update to "server A" (PODEXEC)                   */
/* read a value from "server B" (PODEXEC)                   */
/* send an update to "server B" (PODEXEC)                   */
/* if all requests were successful, commit (SYNCPOINT)      */
/* otherwise roll them back (ROLLBACK)                      */
/* end the transaction (PODTXNE)                            */
/*                                                          */
/****************************/
call Initialize;
call Process_transaction;
call Terminate;
exec cics return;
/****************************/
/* Initialize                                                 */
/*                                                          */
/* Get references to server "A" and server "B".              */
Developing the CICS Two-Phase Commit Client

Example 8: The DATAC Demonstration Module (Sheet 3 of 9)

/* */
/*******************************/

Initialize: PROC;

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

/* Initialize our ORB */
put skip list('Initializing the ORB');
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 */
put skip list('Registering the Interface');
call podreg(addr(Data_interface_interface));
if check_errors('podreg') ^= completion_status_yes then return;

/* Set the pointer to the urlA string. */
call strset(data_url_ptr,
data_urlA,
           length(data_urlA));
if check_errors('strset') ^= completion_status_yes then return;

/* Obtain object A reference from the url. */
call str2obj(data_url_ptr,DataObject_objA);
if check_errors('str2obj') ^= completion_status_yes then return;

/* Releasing the memory. */
call strfree(data_url_ptr);
if check_errors('strfree') ^= completion_status_yes then return;

/* Set the pointer to the urlB string. */
call strset(data_url_ptr,
data_urlB,
           length(data_urlB));
if check_errors('strset') ^= completion_status_yes then return;
/* Obtain object B reference from the url. */
call str2obj(data_url_ptr, DataObject_objB);
if check_errors('str2obj') ^= completion_status_yes then
  return;

/* Releasing the memory. */
call strfree(data_url_ptr);
if check_errors('strfree') ^= completion_status_yes then
  return;
END Initialize;

 domingo 22 de abril de 2001

Example 8: The DATAC Demonstration Module  (Sheet 4 of 9)

  /* Obtain object B reference from the url. */
  call str2obj(data_url_ptr, DataObject_objB);
  if check_errors('str2obj') ^= completion_status_yes then
    return;

  /* Releasing the memory. */
  call strfree(data_url_ptr);
  if check_errors('strfree') ^= completion_status_yes then
    return;

  END Initialize;

  /**************************************************************
  /*                                                                                                                                                                                                 
  /* Process_transaction                                          */
  /**************************************************************
  /* Begin a two-phase commit transaction by calling podtxnb. */
  /* Read a value from "server A". Add 1 to the value and */
  /* update "server A" with the new value. */
  /* Read a value from "server B". Add 1 to the value and */
  /* update "server B" with the new value. */
  /* Check that all requests were successful. */
  /* If so, request a commit by calling SYNCPOINT. */
  /* If not, back out the updates by calling ROLLBACK. */
  /* End the two-phase commit transaction by calling podtxne. */
  /**************************************************************

  Process_transaction: PROC;

  /* Begin a transaction. */
  call podtxnb;
  if check_errors('podtxnb') ^= completion_status_yes then
    return;
  put skip list('Two-phase commit transaction begins');

  call read_value_A;

  if read_result_A = good_result
    then
      do;
        call update_value_A;
      end;
if update_result_A = good_result
  then
    do;
      call read_value_B;
    end;

if read_result_B = good_result
  then
    do;
      call update_value_B;
    end;

if read_result_A = good_result &
  update_result_A = good_result &
  read_result_B = good_result &
  update_result_B = good_result
  then
    do;
      MessageText = 'Two-phase commit transaction completed';
      put skip list('All updates successful -');
      put skip list('request commit');
      call syncpoint;
    end;
  else
    do;
      MessageText = 'A problem was encountered - rolling back';
      put skip list('Some updates were not successful -');
      put skip list('request rollback');
      call rollback;
    end;
/* End the transaction. */
call podtxne;
if check_errors('podtxne') ^= completion_status_yes then
  return;
put skip list('Two-phase commit transaction ends');
exec cics send text from (MessageText length(79) freekb;
END Process_transaction;
***************************************************************/
Example 8: The DATAC Demonstration Module (Sheet 6 of 9)

```c
/* read_value_A */
/* */
/* Read a value from "server A". */
/* */
/***************************************************************************/
read_value_A: PROC;

   call podexec(DataObject_objA,
               read_operation,
               addr(read_operation_args),
               no_user_exceptions);

   if check_errors('podexec') = completion_status_yes then
      do;
         read_result_A = 1;
         put skip list('Successfully read a value from Server A: ');
         put list(read_operation_args.idl_value);
      end;

   END read_value_A;

/***************************************************************************/
/* update_value_A */
/* */
/* Request that "server A" update a value. */
/* */
/***************************************************************************/
update_value_A: PROC;

   write_operation_args.idl_value = read_operation_args.idl_value + 1;
   put skip list('New value for server A: ');
   put list(write_operation_args.idl_value);

   call podexec(DataObject_objA,
               write_operation,
               addr(write_operation_args),
               no_user_exceptions);

   if check_errors('podexec') = completion_status_yes then
      do;
```
Developing the CICS Two-Phase Commit Client

Example 8: The DATAC Demonstration Module (Sheet 7 of 9)

```
update_result_A = 1;
put skip list('Server A has successfully updated the value.');
end;
END update_value_A;

/*************************************************************/
/*                                                           */
/* read_value_B                                              */
/*                                                           */
/* Read a value from "server B".                            */
/*                                                           */
/*************************************************************/
read_value_B: PROC;
call podexec(DataObject_objB,
    read_operation,
    addr(read_operation_args),
    no_user_exceptions);
if check_errors('podexec') = completion_status_yes then
do;
    read_result_B = 1;
    put skip list('Successfully read a value from Server B: ');
    put list(read_operation_args.idl_value);
end;
END read_value_B;

/*************************************************************/
/*                                                           */
/* update_value_B                                            */
/*                                                           */
/* Request that "server B" update a value.                   */
/*                                                           */
/*************************************************************/
update_value_B: PROC;
write_operation_args.idl_value = read_operation_args.idl_value + 1;
put skip list('New value for server B: ');
put list(write_operation_args.idl_value);
```
Example 8: The DATAC Demonstration Module  (Sheet 8 of 9)

call podexec(DataObject_objB,
            write_operation,
            addr(write_operation_args),
            no_user_exceptions);

if check_errors('podexec') = completion_status_yes
then
  do;
    update_result_B = 1;
    put skip list('Server B has successfully updated the value.');
  end;
END update_value_B;

LONGEST_LINES:

END update_value_B;

Example 8: The DATAC Demonstration Module  (Sheet 8 of 9)

/*
   Syncpoint
   */
/* Issue a SYNCPOINT call. */
/*
   */
/***************************************************************************/
SYNCPOINT: PROC;

dcl resp1                         fixed bin(31);
dcl resp2                         fixed bin(31);

exec cics syncpoint
  resp(resp1)
  resp2(resp2);

if resp1 = dfhresp(ROLLEDBACK)
then
  do;
    put skip list('Rollback requested by partner.');
    'Two-phase commit - partner requested a rollback';
  end;
else
  if resp1 ^= dfhresp(NORMAL)
  then
    do;
      put skip list('Syncpoint has failed.');
  end;
else
  do;
      put skip list('Syncpoint has failed.');
  end;
Example 8: The DATAC Demonstration Module (Sheet 9 of 9)

```
END SYNCPOINT;

/*****************************************************************************************/
/*                                                                                     */
/* Rollback                                                                */
/*                                                                                     */
/* Issue a ROLLBACK call.                                                          */
/*                                                                                     */
/*****************************************************************************************/
ROLLBACK: PROC;
  exec cics syncpoint rollback;
END ROLLBACK;

/*****************************************************************************************/
/*                                                                                     */
/* Terminate                                                              */
/*                                                                                     */
/* Release the references to "server A" and "server B".                       */
/*                                                                                     */
/*****************************************************************************************/
Terminate: PROC;
  call objrel(DataObject_objA);
  if check_errors('objrel') ^= completion_status_yes then return;
  call objrel(DataObject_objB);
  if check_errors('objrel') ^= completion_status_yes then return;
  free pod_status_information;
END Terminate;
END DATAC;
```

Explanation of the DATAC module

The CICS DATAC module can be explained as follows:
The CICS DATA module can be explained as follows:

1. `data-urlA` and `data-urlB` define corbaloc URL strings in the `corbaloc:rir` format. These strings identify the servers with which the client is to communicate. The strings can be passed as parameters 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. You can use the `check_errors` function to check 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 to server 'A' is copied.

6. `STR2OBJ` is called to create an object reference to the server 'A' 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:/DataObjectA` (as defined in point 1). See “STR2OBJ” on page 481 for more details of the various forms of corbaloc URL strings and the ways you can use them.

7. `STRSET` is called to create an unbounded string to which the stringified object reference to server 'B' is copied.

8. `STR2OBJ` is called to create an object reference to the server 'B' 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...
Developing the CICS Two-Phase Commit Client

in the form `corbaloc:rir:/DataObjectB` (as defined in point 1). See “STR2OBJ” on page 481 for more details of the various forms of corbaloc URL strings and the ways you can use them.

9. `PODTXNB` is called to indicate the start of two-phase commit processing. The next APPC conversation with the client adapter, which is established at the next call to `PODEXEC`, will be at sync level 2.

10. `PODEXEC` is called in this procedure to read a value from server 'A'.

11. `PODEXEC` is called in this procedure to update a value from server 'A'. Server 'A' will log that an update has been requested, but make no actual changes.

12. `PODEXEC` is called in this procedure to read a value from server 'B'.

13. `PODEXEC` is called in this procedure to update a value from server 'B'. Server 'B' will log that an update has been requested, but make no actual changes.

14. If any call to `PODEXEC` was unsuccessful, ask CICS to initiate rollback processing to undo the updates made by the servers. Server 'A' and 'B' will destroy the log that was holding the potential updates. No actual updates will be made.

15. `PODTXNE` is called to indicate the end of two-phase commit processing. This requests that APPC deallocates the conversation. However, the actual deallocation does not occur until the two-phase commit processing has completed.

16. The CICS transaction calls `SYNCPOINT`. This triggers the start of two-phase commit processing. The client adapter is notified that the CICS transaction has initiated two-phase commit processing. The client adapter requests that server 'A' and server 'B' prepare their updates. Each server replies to the client adapter that they are either able or unable to commit the update. If either server replies that they are unable to commit the update, each server is asked to roll back and destroy the log that was holding the potential update. If both servers reply that they are able to commit the changes, the client adapter requests each server to commit their changes. Two-phase commit processing ends.
Location of the DATAC module

You can find a complete version of the CICS DATAC client module in orbixh1q.DEMOS.CICS.PLI.SRC(DATAC).
Building the Client

JCL to run the Orbix IDL compiler
Before you can build the client, you must run the Orbix IDL compiler on the IDL supplied in orbixhlq.DEMOS.IDL(DATA). Sample JCL to do this can be found in orbixhlq.DEMOS.CICS.PLI.BLD.JCLLIB(DATAIDL).

JCL to build the client
Sample JCL used to compile and link the client can be found in orbixhlq.DEMOS.CICS.PLI.BLD.JCLLIB(DATACB).

Resulting load module
When the JCL has successfully executed, it results in a load module that is contained in orbixhlq.DEMOS.CICS.PLI.LOADLIB(DATACL).
### Building the Servers

#### JCL to run the Orbix IDL compiler
Before you can build the servers, ensure that you have run the Orbix IDL compiler on the IDL supplied in `orbixhlq.DEMOS.IDL(DATA)`. Sample JCL to do this can be found in `orbixhlq.DEMOS.CICS.PLI.BLD.JCLLIB(DATAIDL)`.

**Note:** If you have already built the client, this step should have already been completed.

#### JCL to build the servers
Sample JCL used to compile and link the servers can be found in `orbixhlq.DEMOS.CPP.BLD.JCLLIB(DATASV)`.

#### Resulting load module
When the JCL has successfully executed, it results in a load module that is contained in `orbixhlq.DEMOS.CPP.LOADLIB(DATASV)`.
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:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Define a transaction to CICS.</td>
</tr>
<tr>
<td>2</td>
<td>Provide the CICS client load module to the CICS region.</td>
</tr>
<tr>
<td>3</td>
<td>Start the locator, node daemon, and RRS OTSTM on the server host.</td>
</tr>
<tr>
<td>4</td>
<td>Start the batch servers.</td>
</tr>
<tr>
<td>5</td>
<td>Customize the batch server IORs.</td>
</tr>
<tr>
<td>6</td>
<td>Configure and run the client adapter.</td>
</tr>
</tbody>
</table>

Step 1—Define a transaction to CICS

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

```
DEFINE TRANSACTION (DATC)
GROUP (ORXDEMO)
DESCRIPTION (Orbix Client Two-Phase Commit demo transaction)
PROGRAM (DATACL)
PROFILE (DFHCICSA)
TRANCLASS (DFHTCL00)
DTIMOUT (10)
SPURGE (YES)
TPURGE (YES)
RESSEC (YES)
```
Step 2—Provide client load module to CICS region

Ensure that the orbixhlq.DEMOS.CICS.PLL.LOADLIB 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 DATACL load module to a PDS in the DFHRPL of the relevant CICS region.

Step 3—Start locator, node daemon, and RRS OTSTM on server

This step assumes that you intend running the CICS client against the demonstration batch 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.JCLLIB(LOCATOR).
2. Start the node daemon by submitting orbixhlq.JCLLIB(NODEDAEM).
3. Start the RRS OTSTM server by submitting orbixhlq.JCLLIB(OTSTM).

See “Running the Server and Client” on page 47 for more details of running the locator and node daemon on the batch server host.

See the chapter on Using OTS RRS Transaction Manager in the Mainframe OTS Guide for more details of running the RRS OTSTM server.

Step 4—Start batch servers

This step assumes that you intend running the CICS client against the demonstration batch servers.

Submit the orbixhlq.DEMOS.CPP.RUN.JCLLIB(DATAA) and orbixhlq.DEMOS.CPP.RUN.JCLLIB(DATAB) JCL to start the batch servers.

Step 5—Customize batch server IORs

When you run the demonstration batch servers they publish their IORs to orbixhlq.DEMOS.IORS(DATAA) and orbixhlq.DEMOS.IORS(DATAB).

The demonstration CICS client needs to use these IORs to contact the demonstration batch servers. The demonstration CICS client obtains the object reference for the demonstration batch servers in the form of a corbaloc URL string. A corbaloc URL string can take different formats. For the purposes of this demonstration, the corbalocs take the form corbaloc:rir:/DataObjectA and corbaloc:rir:/DataObjectB.
This form of the corbaloc URL string requires the use of the configuration variables, `initial_references:DataObjectA:reference` and `initial_references:DataObjectB:reference`, in the configuration domain. When you submit the JCL in `orbixhlq.DEMOS.CICS.PLL.BLD.JCLLIB (DATAIORS)`, it automatically adds these configuration entries to the configuration domain:

```
initial_references:DataObjectA:reference = "IOR...";
initial_references:DataObjectB:reference = "IOR...";
```

The IOR values are taken from `orbixhlq.DEMOS.IORS(DATAA)` and `orbixhlq.DEMOS.IORS(DATAB)`. See “STR2OBJ” on page 481 for more details of the various forms of corbaloc URL strings and the ways you can use them.

---

**Step 6—Configure and run client adapter**

The client adapter must now be configured before you can start the client (the 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 `orbixhlq.JCLLIB(CICSCA)`.

**Note:** See “Running a CICS Two-Phase Commit Client against Batch Servers” on page 189 for details of how to run the sample two-phase commit client demonstration.
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:

<table>
<thead>
<tr>
<th>Demonstration</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running a Batch Client against a CICS Server</td>
<td>187</td>
</tr>
<tr>
<td>Running a CICS Client against a Batch Server</td>
<td>188</td>
</tr>
<tr>
<td>Running a CICS Two-PhaseCommit Client against Batch Servers</td>
<td>189</td>
</tr>
</tbody>
</table>
Running a Batch Client against a CICS Server

<table>
<thead>
<tr>
<th>Overview</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steps</td>
<td>The steps to run the demonstration CICS server against the demonstration batch client are:</td>
</tr>
<tr>
<td></td>
<td>1. Ensure that all the steps in “Preparing the Server to Run in CICS” on page 151 have been successfully completed.</td>
</tr>
<tr>
<td></td>
<td>2. Run the batch client as described in “Running the Server and Client” on page 47.</td>
</tr>
<tr>
<td>CICS server output</td>
<td>The CICS server sends the following output to the CICS region:</td>
</tr>
<tr>
<td></td>
<td><code>Simple::call_me invoked</code></td>
</tr>
<tr>
<td>Batch client output</td>
<td>The batch client produces the following output:</td>
</tr>
</tbody>
</table>
|          | `Initializing the ORB`
|          | `Registering the Interface`
|          | `Reading object reference from file`
|          | `invoking Simple::call_me`
|          | `Simple demo complete.` |
## Running a CICS Client against a 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 162 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:

```
Initializing the ORB
Registering the Interface
invoking Simple::call_me
Simple demo complete.
```

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

```
Simple transaction completed
```

### Batch server output
The batch server produces the following output:

```
Initializing the ORB
Registering the Interface
Creating the Object
Writing object reference to file
Giving control to the ORB to process Requests
Simple::call_me invoked
```
Running a CICS Two-Phase Commit Client against Batch Servers

Overview
This subsection describes what you need to do to successfully run the demonstration CICS two-phase commit client against the demonstration batch servers. It also provides an overview of the output produced.

Note: For instructions on recovery processing for any unsuccessful runs of an application, see orbixhlq.DEMOS.CICS.PLI.README(DATAC).

Steps
The steps to run the demonstration CICS two-phase commit client against the demonstration batch servers are:
1. Ensure that all the steps in “Preparing the Client to Run in CICS” on page 183 have been successfully completed.
2. Run the CICS client by entering the transaction name, DATAC, in the relevant CICS region.

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

```
Initializing the ORB
Registering the Interface
Two-phase commit transaction begins
Invoking: read:IDL:Data:1.0
Successfully read a value from server A: 0000000001
New value for server A: 0000000002
Invoking: write:IDL:Data:1.0
Server A has successfully updated the value.
Invoking: read:IDL:Data:1.0
Successfully read a value from server B: 0000000001
New value for server B: 0000000002
Invoking: write:IDL:Data:1.0
Server B has successfully updated the value.
All updates successful - request commit
Two-phase commit transaction ends
```
The CICS client sends the following output to the CICS terminal:

Two-phase commit transaction completed

**Batch server 'A' output**

OTS Recovery Demo Server
Initializing the ORB
Server ID is A
IOR file is DD:IORS(DATAA)
Data file is DD:DATA(DATAA)
Log file is DD:DATA(LOGA)
Resolving TransactionCurrent
Resolving RootPOA
Creating POA with Requires OTS Policy
Creating POA with lifespan policy of Persistent
Creating POA with an ID assignment of USER
Creating Data servant and object
Creating POA for Resource objects
Reading data from file DD:DATA(DATAA)
Value is 1
Writing object reference to DD:IORS(DATAA)
Activation POA for Data object
Data servant read() called
Read-only access: not registering Resource object
Current value is 1
Data servant write() called
Getting coordinator for current transaction
Getting Transaction Identifier
Creating Resource servant
Activating Resource object
Registering Resource object with coordinator
Activating the Resource POA
Setting value to 2
Resource servant prepare() called
Voting to commit the transaction
Writing prepare record
Resource servant commit() called
Writing data to file DD:DATA(DATAA)
Deleting prepare record
Deactivating Resource object
Resource servant destructed
Batch server 'B' produces the following output:

OTS Recovery Demo Server
Initializing the ORB
Server ID is B
IOR file is DD:IORS(DATAB)
Data file is DD:DATA(DATAB)
Log file is DD:DATA(LOGB)
Resolving TransactionCurrent
Resolving RootPOA
Creating POA with REQUIRES OTS Policy
Creating POA with lifespan policy of PERSISTENT
Creating POA with an ID assignment of USER
Creating Data servant and object
Creating POA for Resource objects
Reading data from file DD:DATA(DATAB)
Value is 1
Writing object reference to DD:IORS(DATAB)
Activation POA for Data object
Data servant read() called
Read-only access: not registering Resoure object
Current value is 1
Data servant write() called
Getting coordinator for current transaction
Getting Transaction Identifier
Creating Resource servant
Activating Resource object
Registering Resource object with coordinator
Activating the Resource POA
Setting value to 2
Resource servant prepare() called
Voting to commit the transaction
Writing prepare record
Resource servant commit() called
Writing data to file DD:DATA(DATAB)
deleting prepare record
Deactivating Resource object
Resource servant destructed
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:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL</td>
<td>194</td>
</tr>
<tr>
<td>Modules and Name Scoping</td>
<td>195</td>
</tr>
<tr>
<td>Interfaces</td>
<td>196</td>
</tr>
<tr>
<td>IDL Data Types</td>
<td>213</td>
</tr>
<tr>
<td>Defining Data Types</td>
<td>228</td>
</tr>
</tbody>
</table>
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:

```idl
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 194 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 {
            //...
        };
    };
};
```
In this section

The following topics are discussed in this section:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Contents</td>
<td>198</td>
</tr>
<tr>
<td>Operations</td>
<td>199</td>
</tr>
<tr>
<td>Attributes</td>
<td>201</td>
</tr>
<tr>
<td>Exceptions</td>
<td>202</td>
</tr>
<tr>
<td>Empty Interfaces</td>
<td>203</td>
</tr>
<tr>
<td>Inheritance of Interfaces</td>
<td>204</td>
</tr>
<tr>
<td>Multiple Inheritance</td>
<td>205</td>
</tr>
</tbody>
</table>

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:

```idl
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:

```plaintext
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 {
    [member1;]...
};
```

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 {};  // Exception for insufficient funds
        void withdraw(in CashAmount amount) raises (InsufficientFunds); //...  // Raises InsufficientFunds exception when withdrawal fails
    };
};
```
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:

```plaintext
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 6 shows the inheritance hierarchy for the Account interface, which is defined in “Multiple inheritance IDL sample” on page 205.
Figure 6: 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:

```idl
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:

```idl
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:

```idl
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 for account already exists
        exception AccountNotFound (AccountId account_id); // Exception for account not found

        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.
Valuetyes

Overview

Valuetyes 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 valuetyes.
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:

<table>
<thead>
<tr>
<th>Data Type Category</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built-in Data Types</td>
<td>214</td>
</tr>
<tr>
<td>Extended Built-in Data Types</td>
<td>217</td>
</tr>
<tr>
<td>Complex Data Types</td>
<td>220</td>
</tr>
<tr>
<td>Enum Data Type</td>
<td>221</td>
</tr>
<tr>
<td>Struct Data Type</td>
<td>222</td>
</tr>
<tr>
<td>Union Data Type</td>
<td>223</td>
</tr>
<tr>
<td>Arrays</td>
<td>225</td>
</tr>
<tr>
<td>Sequence</td>
<td>226</td>
</tr>
<tr>
<td>Pseudo Object Types</td>
<td>227</td>
</tr>
</tbody>
</table>

## 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.
Table 20 shows a list of CORBA IDL built-in data types (where the \( \leq \) symbol means 'less than or equal to').

**Table 20: Built-in IDL Data Types, Sizes, and Values**

<table>
<thead>
<tr>
<th>Data type</th>
<th>Size</th>
<th>Range of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>( \leq 16 ) bits</td>
<td>(-2^{15}...2^{15}-1)</td>
</tr>
<tr>
<td>unsigned short (^a)</td>
<td>( \leq 16 ) bits</td>
<td>(0...2^{16}-1)</td>
</tr>
<tr>
<td>long</td>
<td>( \leq 32 ) bits</td>
<td>(-2^{31}...2^{31}-1)</td>
</tr>
<tr>
<td>unsigned long (^b)</td>
<td>( \leq 32 ) bits</td>
<td>(0...2^{32}-1)</td>
</tr>
<tr>
<td>float</td>
<td>( \leq 32 ) bits</td>
<td>IEEE single-precision floating point numbers</td>
</tr>
<tr>
<td>double</td>
<td>( \leq 64 ) bits</td>
<td>IEEE double-precision floating point numbers</td>
</tr>
<tr>
<td>char</td>
<td>( \leq 8 ) bits</td>
<td>ISO Latin-1</td>
</tr>
<tr>
<td>string</td>
<td>Variable length</td>
<td>ISO Latin-1, except NUL</td>
</tr>
<tr>
<td>string(&lt;\text{bound}&gt;) (^c)</td>
<td>Variable length</td>
<td>ISO Latin-1, except NUL</td>
</tr>
<tr>
<td>boolean</td>
<td>Unspecified</td>
<td>TRUE or FALSE</td>
</tr>
<tr>
<td>octet</td>
<td>( \leq 8 ) bits</td>
<td>0x0 to 0xff</td>
</tr>
<tr>
<td>any</td>
<td>Variable length</td>
<td>Universal container type</td>
</tr>
</tbody>
</table>

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

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

\(^c\) 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 20 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 21 shows a list of CORBA IDL extended built-in data types (where the \( \leq \) symbol means 'less than or equal to').

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>long long(^a)</td>
<td>( \leq 64 ) bits</td>
<td>(-2^{63} \ldots 2^{63}-1)</td>
</tr>
<tr>
<td>unsigned long long(^a)</td>
<td>( \leq 64 ) bits</td>
<td>(0 \ldots -2^{64}-1)</td>
</tr>
<tr>
<td>long double(^b)</td>
<td>( \leq 79 ) bits</td>
<td>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.</td>
</tr>
<tr>
<td>wchar</td>
<td>Unspecified</td>
<td>Arbitrary codesets</td>
</tr>
<tr>
<td>wstring</td>
<td>Variable length</td>
<td>Arbitrary codesets</td>
</tr>
<tr>
<td>fixed(^c)</td>
<td>Unspecified</td>
<td>(\leq 31) significant digits</td>
</tr>
</tbody>
</table>

\(^{a}\) 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, \(0 \ldots 2^{64}-1\)).

\(^{b}\) 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, \(\leq 64\) bits).

\(^{c}\) Due to compiler restrictions, the PL/I range of values for the fixed type is \(\leq 15\) 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:

```idl
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:

```idl
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 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:

```idl
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:

```idl
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:

```idl
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:

```idl
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 \textit{dimension-spec} must be a non-zero positive constant integer expression):

\[ \text{typedef} \ element-type \ array-name \ [\text{dimension-spec}] \ldots \]

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:

\texttt{typedef Account portfolio[MAX_ACCT_TYPES][MAX_ACCTS]}

\textbf{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:

```idl
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:

```idl
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:

<table>
<thead>
<tr>
<th>Constants</th>
<th>page 229</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant Expressions</td>
<td>page 232</td>
</tr>
</tbody>
</table>

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:

```idl
module BankDemo {

    interface Account {
        //...
    }

    typedef Account StandardAccount;
}
```
Definitions Data Types

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 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 9:** 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
```
Wide character and string constants

Wide character and string constants use C++ syntax. Use universal character codes to represent arbitrary characters. For example:

```c
const char C11 = '\';      // backslash
const char C12 = '\?';      // question mark
const char C13 = '\'';      // single quote
```

// String constants support the same escape sequences as C++

```c
const string S1 = "\"";      // 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'
```

Example 9: List of character constants (Sheet 2 of 2)

Boolean constants

Boolean constants use the `FALSE` and `TRUE` keywords. Their use is unnecessary, inasmuch as they create unnecessary aliases:

```c
// 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.

```c
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:

```cpp
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:

```cpp
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:

```c++
// 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.

```c++
// You can use bitwise operators to define constants.
const long ALL_ONES = -1;       // 0xffffffff
const long LHW_MASK = ALL_ONES << 16; // 0xffffff00
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.
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:

<table>
<thead>
<tr>
<th>Mapping for Identifier Names</th>
<th>page 237</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping Very Long and Leading Underscored Names</td>
<td>page 239</td>
</tr>
<tr>
<td>Mapping for Basic Types</td>
<td>page 241</td>
</tr>
<tr>
<td>Mapping for Boolean Type</td>
<td>page 245</td>
</tr>
<tr>
<td>Mapping for Enum Type</td>
<td>page 246</td>
</tr>
<tr>
<td>Mapping for Octet and Char Types</td>
<td>page 247</td>
</tr>
<tr>
<td>Mapping for String Types</td>
<td>page 248</td>
</tr>
<tr>
<td>Mapping for Fixed Type</td>
<td>page 251</td>
</tr>
<tr>
<td>Mapping for Struct Type</td>
<td>page 254</td>
</tr>
</tbody>
</table>
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 193 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 323 for more details):

module_name_interface_name_IDL_variable_name

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 239.
- 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 293 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 242 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 237, 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:

```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:

```plaintext
dcl 1 BankLoans_Mortgages_c_e9c_args aligned like
    BankLoans_Mortgages_c_e9c_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 22 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 22: Mapping for Basic IDL Types

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>CORBA Typedef Name</th>
<th>PL/I Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>CORBA-short</td>
<td>FIXED BIN(15)</td>
</tr>
<tr>
<td>long</td>
<td>CORBA-long</td>
<td>FIXED BIN(31)</td>
</tr>
<tr>
<td>unsigned short</td>
<td>CORBA-unsigned-short</td>
<td>FIXED BIN(15)⁹</td>
</tr>
<tr>
<td>unsigned long</td>
<td>CORBA-unsigned-long</td>
<td>FIXED BIN(31)⁹</td>
</tr>
<tr>
<td>float</td>
<td>CORBA-float</td>
<td>FLOAT DEC(6)</td>
</tr>
<tr>
<td>double</td>
<td>CORBA-double</td>
<td>FLOAT DEC(16)</td>
</tr>
<tr>
<td>char</td>
<td>CORBA-char</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>boolean</td>
<td>CORBA-boolean</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>octet</td>
<td>CORBA-octet</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>enum</td>
<td>CORBA-enum</td>
<td>FIXED BIN(31)⁹,b</td>
</tr>
<tr>
<td>fixed&lt;d,s&gt;</td>
<td>Fixed&lt;d,s&gt;</td>
<td>FIXED DEC(d,s)</td>
</tr>
<tr>
<td>any</td>
<td>CORBA-any</td>
<td>See “Mapping for the Any Type” on page 262.</td>
</tr>
<tr>
<td>long long</td>
<td>CORBA-long-long</td>
<td>FIXED BIN(xx)⁹</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>CORBA-unsigned-long-long</td>
<td>FIXED BIN(xx)⁹,b,c</td>
</tr>
</tbody>
</table>

⁹ Includes flags for sign.

b Fixed-point format.

c On This Platform.
Example

The example can be broken down as follows:

1. Consider the following IDL, stored in an IDL member called EXAMPLE:

```idl
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";
}
```

Table 22: Mapping for Basic IDL Types

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>CORBA Typedef Name</th>
<th>PL/I Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>wchar</td>
<td>CORBA-wchar</td>
<td>GRAPHIC</td>
</tr>
</tbody>
</table>

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 for the PL/I representation of an enum is 31 bits.

c. The maximum number of digits allowed in a **FIXED BIN** is 31 bits if you are using the IBM PL/I for MVS & VM compiler, or 63 bits if you specify the -E option with the Orbix IDL compiler and are using a version of the IBM Enterprise PL/I for z/OS compiler.
2. The preceding IDL maps to the following in the idlmembername include member:

```idl
module extras {
    const long  elong = 760224;
};
```

```idl
/*-----------------------------------------------*/
/* 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 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 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);
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:

   ```plaintext
   interface example {
       attribute boolean full;
   };
   ```

2. The preceding IDL maps to the following PL/I:

   ```plaintext
   /* 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);
   ```
CHAPTER 6 | IDL-to-PL/I Mapping

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:
   
   ```idl
   interface weather {
     enum temp {cold, warm, hot};
   };
   ```

2. The preceding IDL maps to the following PL/I:
   
   ```pli
   /*----------------------------------------------------------------------*/
   /* 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;
   /*----------------------------------------------------------------------*/
   /*----*/
   /*----*/
   if todays_temp = weather_temp_cold then
     put skip list('Brr, it is cold outside!');
   /*----------------------------------------------------------------------*/
   ```

3. It can be used as follows:
   
   ```pli
   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 242 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 242 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
//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:

```pli
/* 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:

```pli
/* 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 \texttt{STRSETS} function, with the same argument signature, to copy the specified number of characters, including trailing spaces.

\textbf{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:

   \begin{verbatim}
   interface example {
       attribute string mystring;
       string<10>getname(in string code);
   };
   \end{verbatim}

2. The Orbix IDL compiler generates the following PL/I, based on the preceding IDL:

   \begin{verbatim}
   dcl 1 example_mystring_type based,
   3 result                  ptr       init(sysnull());
   
   dcl 1 example_getname_type based,
   3 code                    ptr       init(sysnull()),
   3 result                  char(10)  init('');
   \end{verbatim}
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:

```plaintext
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
//IDL
interface example {
    typedef fixed<5,2>       typesal;
    typedef 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):

```pli
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:

```plaintext
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:

```plaintext
Salary = 165.78
TaxRate = 0.9876
Millions = 23F+6
Small = 0.004
```

**Note:** The maximum number of figures (not significant digits) allowed is 15 if you do not specify the `-E` option with the Orbix IDL compiler and you are using the IBM PL/I for MVS & VM compiler. However, this is not an issue if you specify the `-E` option with the Orbix IDL compiler and you are using any version of the IBM Enterprise PL/I for z/OS compiler.
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:

```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:

```pli
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:

   ```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 `idlmembernameT` include member:

   ```pli
   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 `idlmembernameT` include member:

   ```pli
   dcl 1 example_test_attr aligned like example_test_type;
   ```
The union branches are generated as follows in the \texttt{idlmembername}\texttt{include} member:

```c
/*---------------------------------------------*/
/* 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);

select (example_test_attr.d) 
  when (1) 
    put skip (Value of case_1 is: ', example_test_result_case_1);
  when (2) 
    put skip (Value of case_2 is: ', example_test_result_case_2);
  otherwise 
    put skip (Value of def_case is: ', example_test_result_def_case);
end;
```

Compiler restrictions

Because the PL/I for MVS & VM compiler does not support unions directly, the union branches (in the preceding example, \texttt{case_1}, \texttt{case_2}, and \texttt{def_case}) are declared separately from the union structure. The union branches use the storage defined by the \texttt{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 \texttt{case_2}, which is a \texttt{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 \texttt{get} on the attribute, you can use a \texttt{select} statement as follows:
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`
server implementation module):

```plaintext
/*------------------------------------------------------------*/
/* 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 \texttt{SEQMAX}), the length of the sequence in elements (accessed via \texttt{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:

```idl
interface example {
  typedef              sequence<long, 10>  seqlong10;
  attribute            seqlong10 myseq;
};
```

The preceding IDL maps to the following PL/I:

```pli
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:

```idl
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:

```pli
/* 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 `idimembername include member`):

```pli

dcl seq_size       fixed bin(31)   init(20);
dcl 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:

```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 idlmembernameT include member:

```pli
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 idlmembernameT include member:

```pli
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:

```pli
example_long_array_args.result(1,3) = 22;
```
CHAPTER 6 | IDL-to-PL/I Mapping

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:

```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 include member:

```pli
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 389 and “TYPEGET” on page 486 for details and an example of how to access the contents of an any. See “ANYSET” on page 391 and “TYPESET” on page 489 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:

```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` include member:

```pli
dcl 1 example_addName_type based,
    3 idl_name            ptr            init(sysnull());
```
3. The Orbix IDL compiler generates the following code in the
idlmembername include member:

```idl
/*-------------------------------------------------------*/
/* 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,
    exception_id         ptr,
    d                    fixed bin(31)  init(0),
    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),
    value1               fixed bin(31)  init(0),
    reason               char(32)       init('');
dcl 1 example_worse_exc    based(EXAMPLE_user_exceptions.u),
    value2               fixed bin(15)  init(0),
    errorcode            char(16)       init(''),
    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` 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` 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.

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:

```plaintext
/* 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`:

```plaintext
Example 10: Client Code to Test Exception (Sheet 1 of 2)

/* Call podexec to perform operation addName. */
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 10: Client Code to Test Exception  (Sheet 2 of 2)

```plaintext
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;
end;
```
Mapping for Typedefs

Overview

This section describes how typedefs are mapped to PL/I.

IDL-to-PL/I mapping for typedefs

Typedefs 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:

```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, \textit{stru}, and for the attribute structure; however, it does not generate a based identifier for the fixed type. The based variables for the struct, \textit{stru}, are generated in the \textit{idlmembernameT} include member as follows:

```idl
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 \textit{idlmembernameM} include member, which makes use of the attribute's based structure:

```idl
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 `idlmembernameT` 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 `idlmembernameM` 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 `idlmembernameT` include member, and with `_args` in the `idlmembernameM` include member, for example:

   ```pli
   dcl 1 my_operation_type_based, 
       3 my_argument fixed bin(31) init(0); 
   ...
   ```

2. A declaration is generated in the `idlmembernameT` 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:

   ```pli
   dcl test_my_operation char(36) 
       init('my_operation:IDL:test:1.0 ');
   ```

3. The operation declaration is also used in the `idlmembernameD` 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 idlmembername1 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 idlmembername1 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 `idlmembernameM` include member:

```plaintext
/*-------------------------------------------------------*/
/* 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;

/*-------------------------------------------------------*/
/* 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 ');
```

4. The following is generated in the `idlmembernameT` include member:

```plaintext
/*-------------------------------------------------------*/
/* 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` include member:

```plaintext
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` member:
CHAPTER 6 | IDL-to-PL/I Mapping

/*-------------------------------------------------------*/
/* 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 _idlmembernameT_ 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 _idlmembernameT_ 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 _idlmembernameM_ include member.

   The top-level identifier (that is, at dcl 1 level) for each attribute declaration is suffixed with _type_ in the _idlmembernameT_ include member, and with _attr_ in the _idlmembernameM_ include member (that is, _FQN_attributename_type_ and _FQN_attributename_attr_).
2. Two declarations are generated in the `idlmembernameI 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 a server. For example, the following is an example of two declarations based on the `myshort` attribute in the `example` interface:

```idl
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 `idlmembernameD 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 `idlmembernameI 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:

```idl
interface example
{
    readonly attribute long mylong;
    attribute short myshort;
}
```
2. Based on the preceding IDL, the following attribute structures are generated in the `idlmembernameT` include member:

```c
/*-------------------------------------------------------*/
/* 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);
/*-------------------------------------------------------*/
/* 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 example_myshort_type;
```

3. Based on the preceding IDL, the following attribute structures are generated in the `idlmembernameT` include member:

```c
/*-------------------------------------------------------*/
/* 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);
```
4. The following is generated in the \texttt{idlmembername}T include member:

```idl
/*-------------------------------------------------------*/
/* 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 ');
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;
```

5. The following \texttt{select} statement is also generated in the \texttt{idlmembernameD} include member:

```idl
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;
```
6. The following skeleton procedures are generated in the
\texttt{idlmembername:} include member:

```idl
when (example_set_mylong) do;
    call podget(addr(example_mylong_attr));
    if check_errors('podget') ^= completion_status_yes
        then return;

    call proc_example_set_mylong
        (addr(example_mylong_attr));

    call podput(addr(example_mylong_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;
```

```idl
/*-------------------------------------------------------*/
/* 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;
```
CHAPTER 6 | IDL-to-PL/I Mapping

`/*-----------------------------------------------*/
/* 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 EXAMPLEI;`
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:

   ```idl
   interface example
   {
     void myoperation();
   };
   ```

2. The preceding IDL maps to the following PL/I:

   ```pli
   /*-----------------------------------------------------------------------*/
   /* 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.
CHAPTER 6 | IDL-to-PL/I Mapping

Mapping for Inherited Interfaces

Overview

This section describes how inherited interfaces are mapped to PL/I.

Note: From Orbix 6.2 onwards, the IDL-to-PL/I plug-in no longer generates typedefs for inherited types by default. This is because typedefs generated for the base class are the same as those for any inherited class. Use the -Li option if you want to generate typedefs for inherited typedefs for the purposes of backwards compatibility with code generated by previous versions of the IDL-to-PL/I plug-in.

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:

   ```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 include member:

   ```pli
   select(operation);
   when (Account_get_mybaseshort) do;
   ```

Example 11: The idlmembername Example (Sheet 1 of 4)
Example 11: The idlmembernameD Example (Sheet 2 of 4)

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 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;
Example 11: The idlmembernameD Example  (Sheet 2 of 4)

```idl
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
     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
```

Example 11: The idlmembernameD Example  (Sheet 2 of 4)
Example 11: The idlmembernameD Example (Sheet 2 of 4)

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
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
Example 11: The idlmembernameD Example  (Sheet 3 of 4)

```idl
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));

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_mybaseshort
(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_mybaseshort
(addr(SavingAccount_mybaseshort_attr));
```
Example 11: The idlmembernameT Example  (Sheet 4 of 4)

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;

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 idlmembernameT include
member:

Example 12: The idlmembernameT Example  (Sheet 1 of 3)

/*-------------------------------------------------------*/
/* Interface: */
/* Account */
/* */
/* Mapped name: */
/* Account */
/* */
/* Inherits interfaces: */
/* (none) */
/*-------------------------------------------------------*/
/* Attribute: mybaseshort */
/* Mapped name: mybaseshort */
/* Type: short (read/write) */
/* */
/*-------------------------------------------------------*/
Example 12: The idlmembernameT Example (Sheet 2 of 3)

dcl 1 Account_mybaseshort_type based,
    3 result                   fixed bin(15)   init(0);
/*-------------------------------------------------------*/
/* Attribute:       mybasefunc                           */
/* Mapped name:     mybasefunc                           */
/* Arguments:       <in> long mybaselong                 */
/* Returns:         void                                 */
/* User Exceptions: none                                 */
/* ------------------------------------------------------*/
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);
/*-------------------------------------------------------*/
/* 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 List section                                */
/* Contains a list of the interface's operations and     */
/* attributes.                                           */
Example 12: The idlmembernameT Example  (Sheet 3 of 3)

```c
/*-------------------------------------------------------*/
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 ');
*/
```
CHAPTER 6 | IDL-to-PL/I Mapping

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:

```idl
interface example1
{
  readonly attribute long mylong;
};

interface example2
{
  readonly attribute long mylong;
};
```
2. The \textit{idlmembername1} member includes \textit{idlmembername2}, to determine which server operation procedure is to be called. For example:

```pascal
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 z/OS 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:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running the Orbix IDL Compiler</td>
<td>294</td>
</tr>
<tr>
<td>Generated PL/I Source and Include Members</td>
<td>302</td>
</tr>
<tr>
<td>Orbix IDL Compiler Arguments</td>
<td>305</td>
</tr>
<tr>
<td>Orbix IDL Compiler Configuration</td>
<td>331</td>
</tr>
</tbody>
</table>

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 z/OS. If you are using z/OS UNIX System Services, references to z/OS 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 z/OS UNIX System Services.

In this section
This section discusses the following topics:

- Running the Orbix IDL Compiler in Batch page 295
- Running the Orbix IDL Compiler in UNIX System Services page 300
## 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 295.
- “Running the Orbix IDL compiler” on page 295.
- “Example of the batch SIMPLIDL JCL” on page 296.
- “Description of the JCL” on page 297.
- “Obtaining IDL in batch” on page 297.
- “ORXCOPY” on page 299.

### 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.JCLLIB(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:

```plaintext
//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.PROCLIB)
// INCLUDE MEMBER=(ORXVARS)
//*
//IDLPLI EXEC ORXIDL,
// SOURCE=SIMPLE,
// IDL=ORBIX..DEMOS.IDL,
// COPYLIB=ORBIX..DEMOS.PLI.PLINCL,
// IMPL=ORBIX..DEMOS.PLI.SRC,
// IDLPARM='-pli:-V'
```

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 idlmembernameV server mainline source code member. See “Orbix IDL Compiler Arguments” on page 305 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.JCLLIB(SIMPLIDL). For details of the JCL for the CICS or IMS version of the demonstration see “Example of the SIMPLIDL JCL” on page 62 or “Example of the SIMPLIDL JCL” on page 134.
Running the Orbix IDL Compiler

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.PROCLIB(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 308 for details of how to specify the Orbix IDL compiler arguments as parameters to it.

Obtaining IDL in batch

In batch, IDL resides in a data set or PDS member with the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record format</td>
<td>Fixed block (FB)</td>
</tr>
<tr>
<td>Record length</td>
<td>80</td>
</tr>
<tr>
<td>Block size</td>
<td>27920 (the block size might vary, depending on your installation standards)</td>
</tr>
</tbody>
</table>

Each record in the data set or PDS member should not exceed 71 characters. If a record is longer than 71 characters, the record must be continued into the next record, as follows:

- Code the IDL record up to column 71.
- Put the "\" continuation character in column 72.
- Continue the IDL record beginning in column 1 of the next record.
The following is an example of the preceding points:

```idl
module Banking
{
    typedef float CashAmount; //Define a named type to represent money
}
```

When IDL is brought into the batch environment from another environment, such as Windows or UNIX, the records in the IDL might be longer than 71 characters. To avoid having to edit the IDL manually to conform to the continuation rules, use the following procedure to obtain IDL in batch:

1. Allocate a data set with the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record format</td>
<td>Variable Blocked (VB)</td>
</tr>
<tr>
<td>Record length</td>
<td>2052</td>
</tr>
<tr>
<td>Block size</td>
<td>27998</td>
</tr>
<tr>
<td>Space allocation units</td>
<td>Tracks</td>
</tr>
<tr>
<td>First extent</td>
<td>1</td>
</tr>
<tr>
<td>Second extent</td>
<td>1</td>
</tr>
<tr>
<td>Data set type</td>
<td>Physical Sequential (PS)</td>
</tr>
</tbody>
</table>

2. Use File Transfer Protocol (FTP) to copy the IDL from Windows or UNIX to this data set.

3. Run the ORXCOPY program to copy the IDL from the data set in point 2 into the IDL data set or PDS member. ORXCOPY automatically formats each line of IDL that is greater than 71 characters.
The following is an example of ORXCOPY:

```
//JOBNAME JOB ...
/*
   JCLLIB ORDER=(orbixhlq.ORBIX62.PROCLIB)
   INCLUDE MEMBER=(ORXVARS)
/*
/* Copy from a variable-length record IDL file into
/* the (fixed-length record) IDL file. Long
/* lines will be split across records with a
/* backslash.
/*
//GO EXEC PROC=ORXG,
// PROGRAM=ORXCOPY,
// PPARM='DD:IN DD:OUT (LONG)'
//IN DD DISP=SHR, DSN=&ORBIX..LONG.IDL
//OUT DD DISP=SHR, DSN=&ORBIX..DEMOS.IDL
```

The preceding code can be explained as follows:

1. The ORXCOPY program is used to copy the IDL from a variable length data set into a fixed length PDS with long lines correctly formatted for continuation.

2. &ORBIX..LONG.IDL is a variable length data set that contains IDL that has been copied from Windows or UNIX via FTP.

3. &ORBIX..DEMOS.IDL is a fixed length PDS. The IDL is copied from the variable length data set into the PDS member called LONG. Any line that was originally longer than 71 characters is properly formatted for continuation onto the next line.
Running the Orbix IDL Compiler in UNIX System Services

Overview

This subsection describes how to run the Orbix IDL compiler in z/OS UNIX System Services. It discusses the following topics:

- “Orbix IDL compiler configuration” on page 300.
- “Prerequisites to running the Orbix IDL compiler” on page 300.
- “Running the Orbix IDL compiler” on page 295.

Note: Even though you can run the Orbix IDL compiler in z/OS UNIX System Services, Orbix does not support subsequent building of Orbix PL/I applications in z/OS 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
idlfilenameV server mainline source code file. See “Orbix IDL Compiler
Arguments” on page 305 for more details of Orbix IDL compiler arguments.
See “Generated PL/I Source and Include Members” on page 302 and “Orbix
IDL Compiler Configuration” on page 331 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 23 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>
<thead>
<tr>
<th>Member Name</th>
<th>Member Type</th>
<th>Compiler Argument Used to Generate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idlmembernameI</td>
<td>Source code</td>
<td>-S</td>
<td>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.</td>
</tr>
<tr>
<td>idlmembernameV</td>
<td>Source code</td>
<td>Generated by default</td>
<td>This is the server mainline source code member. It is generated by default unless you specify the -V argument to prevent generation of it.</td>
</tr>
<tr>
<td>idlmembernameO</td>
<td>Include member</td>
<td>Generated by default</td>
<td>This is the select include member. It contains a select statement that determines the appropriate implementation function for the attribute or operation being called.</td>
</tr>
<tr>
<td>idlmembernameL</td>
<td>Include member</td>
<td>Generated by default</td>
<td>This is the alignment include member. It contains procedures to perform the PL/I alignment calculations on behalf of the PL/I runtime.</td>
</tr>
<tr>
<td>idlmembernameM</td>
<td>Include member</td>
<td>Generated by default</td>
<td>This is the main include member. It stores all the PL/I structures and declarations.</td>
</tr>
</tbody>
</table>
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. On native z/OS, member names are always generated in uppercase. On z/OS UNIX System Services, filenames are generated in lowercase by default. However, you can use the AllCapsFilenames configuration variable or the -lc Orbix IDL compiler argument on z/OS UNIX System Services, to generate filenames in uppercase instead. (Generating filenames in uppercase on z/OS UNIX System Services does not affect the case of file extensions however.)
Filename extensions on z/OS UNIX System Services

If you are running the Orbix IDL compiler in z/OS 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 332 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:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary of the arguments</td>
<td>306</td>
</tr>
<tr>
<td>Specifying Compiler Arguments</td>
<td>308</td>
</tr>
<tr>
<td>-D Argument</td>
<td>311</td>
</tr>
<tr>
<td>-E Argument</td>
<td>312</td>
</tr>
<tr>
<td>-L Argument</td>
<td>314</td>
</tr>
<tr>
<td>-M Argument</td>
<td>316</td>
</tr>
<tr>
<td>-O Argument</td>
<td>323</td>
</tr>
<tr>
<td>-S Argument</td>
<td>325</td>
</tr>
<tr>
<td>-T Argument</td>
<td>326</td>
</tr>
<tr>
<td>-V Argument</td>
<td>329</td>
</tr>
<tr>
<td>-W Argument</td>
<td>330</td>
</tr>
</tbody>
</table>
Summary of the arguments

Overview

For the purposes of Orbix PL/I application development, the Orbix IDL compiler arguments can be categorized as follows:

- General arguments not specific to the PL/I plug-in.
- Arguments specific to the PL/I plug-in and qualified by the `-pli` switch.

Note: Orbix IDL compiler arguments relating to other plug-ins are also available, such as those concerned with COBOL, C++ or Java application development. See the relevant Orbix programmer’s guides for those languages for more details of their associated arguments.

This subsection provides an introductory overview of both the general Orbix IDL compiler arguments and those that are specific to the PL/I plug-in. Each argument that is specific to the PL/I plug-in is explained in more detail further on in this section.

Summary of general arguments

The general Orbix IDL compiler arguments can be summarized as follows:

- `-D name [=value]` Defines the preprocessor's name.
- `-E` Runs preprocessor only, prints on stdout.
- `-I dir` Includes `dir` in search path for preprocessor.
- `-N` Generates code for `#include` files.
- `-U name` Undefines name for preprocessor.
- `-u` Prints usage message and exits.q
- `-V` Print version information and exits.
- `-v` Traces compilation stages.
- `-w` Suppresses IDL compiler warning messages.
- `-flags` Lists all valid arguments to the Orbix IDL compiler.

Note: All these arguments are optional. This means that they do not have to be specified as parameters to the Orbix IDL compiler.
Summary of PL/I plug-in arguments

The Orbix IDL compiler arguments that are specific to the PL/I plug-in 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 z/OS UNIX System Services only.

- `-E` Enable Enterprise PL/I extended precision for long long types and 31-digit fixed types.

- `-L` Limit code generation / legacy options.

- `-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` Generate message code as either `put skip` or `display`.

  **Note:** 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 z/OS UNIX System Services. It discusses the following topics:

● “Specifying general compiler arguments” on page 308.
● “Specifying compiler arguments in UNIX System Services” on page 309.

Specifying general compiler arguments
General compiler arguments are those listed in “Summary of general arguments” on page 306. These arguments must be separated by spaces. For example:

-Dname[=value] -E -Idir -N -Uname -u -V -v -w

Specifying PL/I plug-in arguments
Compiler arguments specific to the PL/I plug-in are those listed in “Summary of PL/I plug-in arguments” on page 307. They must be qualified with the -pli switch. Each of these arguments must be preceded by a colon (that is, ":"), and there must be no spaces between any characters or any arguments. For example:

-pli[:=D[option][dir]][:=E[option]] [:=N[option]] [:=M[option][typename]] [:=O[typename]] [:=S] [:=T[option]] [:=V] [:=W[option]]
Specifying both general and PL/I plug-in arguments

You can specify both general and PL/I plug-in arguments together as parameters to the Orbix IDL compiler. It does not matter whether you specify general arguments first or PL/I plug-in arguments first. The main thing to remember is that:

• General arguments must be separated with spaces.
• PL/I plug-in arguments must be qualified with the -pli switch, must be preceded by a colon, and must not include any spaces between any characters or arguments.

In the following example, general arguments are specified first, followed by PL/I plug-in arguments:

```bash
-Dname=value -E -Idir -N -Uname -u -V -v -w -pli[:D[option][dir]][:-E][:-L[option]]
[:M[option][membername]] [:Gmembername][:-S][:-T[option]][:-V][:-W[option]]'
```

Specifying compiler arguments in batch

On native z/OS, 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 the Orbix IDL compiler. The parameters for the IDLPARM entry in the JCL take the following format:

```bash
// IDLPARM='"-Dname=value -E -Idir -N -Uname -u -V -v -w -pli[:D[option][dir]][:-E]
[:L[option]][:-M[option][membername]][:-Gmembername][:-S][:-T[option]][:-V]
[:W[option]]"'
```

See “Running the Orbix IDL Compiler” on page 294 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.

Specifying compiler arguments in UNIX System Services

The parameters to the Orbix IDL compiler take the following format on z/OS UNIX System Services:

```bash
-Dname=value -E -Idir -N -Uname -u -V -v -w -pli[:D[option][dir]][:-E][:-L[option]]
[:M[option][membername]][:-Gmembername][:-S][:-T[option]][:-V][:-W[option]]'
```
Specifying default PL/I plug-in arguments

It is possible to enable the Orbix IDL compiler to process PL/I plug-in arguments by default, without having to specify those arguments when running the Orbix IDL compiler. You can do this via settings in the pli scope of the orbixhlq.CONFIG(IDL) configuration member. See “Orbix IDL Compiler Configuration” on page 331 for more details.
-D Argument

Overview

By default, when you run the Orbix IDL compiler in z/OS UNIX System Services, it generates source code and include files into the current working directory. You can use the PL/I plug-in argument, -D, with the Orbix IDL compiler to redirect some or all of the generated output into alternative directories.

Note: The PL/I plug-in argument, -D, is relevant only if you are running the Orbix IDL compiler on z/OS UNIX System Services. It is ignored if you specify it when running the Orbix IDL compiler on native z/OS.

Specifying the -D argument

The -D PL/I plug-in 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:-McreateMyfile.map myfile.idl
```

Note: See the rest of this section for more details of how to generate source code and IDL mapping files.
-E Argument

Overview

The PL/I plug-in argument, -E, enables extended precision support in the IDL PL/I backend, for use by the Enterprise PL/I compiler. As shown in Table 24, it provides support for true long long and 31-digit fixed types.

Table 24: CORBA Type Support Provided by -E Option

<table>
<thead>
<tr>
<th>CORBA Type</th>
<th>Without -E Option</th>
<th>With -E Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>(unsigned) long long fixed bin(31)</td>
<td>fixed bin(31)</td>
<td>fixed bin(63)</td>
</tr>
<tr>
<td>maximum fixed type precision</td>
<td>up to 15 digits</td>
<td>up to 31 digits</td>
</tr>
</tbody>
</table>

Note: As well as the above, the -E argument, when used with -TIMS, alters the default list of include files within the server's mainline, to include the IMSPCBE include file instead of IMSPCB. The IMSPCBE include file is identical to IMSPCB, except that it also includes the assignable keyword in the PCBLIST struct, to allow the Enterprise PL/I compiler to compile programs that include this file when the option DEFAULT(NONASSIGNABLE) is passed to the PL/I compiler.

Example

For example, consider the following IDL:

```idl
interface extended_test
{
  typedef fixed<31,0> Fixed_31_0;
  attribute Fixed_31_0 my_fixed;

  attribute long long mylonglong;
  attribute unsigned long long myulonglong;
};
```
By default (that is, without the use of the `–E` PL/I plug-in argument), the preceding IDL would be mapped to the following PL/I:

```IDL
  dcl 1 extended_test_my_fixed_type based,
        3 result                    fixed dec(15,0) init(0);
  dcl 1 extended_test_mylonglong_type based,
        3 result                    fixed bin(31) init(0);
  dcl 1 extended_test_myulonglong_type based,
        3 result                    fixed bin(31) init(0);
```

Additionally, if the IDL contains long long or 31-digit fixed types but you do not specify the `–E` PL/I plug-in argument, the Orbix IDL compiler issues the following warning:

```IDL
  idl: "DD:IDLIN(EXTENDED)", line 3: Warning: Unsupported Type,
       Fixed type argument too large - field size truncated.
```

Alternatively, if the IDL contains long long or 31-digit fixed types and you do specify the `–E` PL/I plug-in argument, the Orbix IDL compiler generates the following PL/I based on the preceding IDL:

```IDL
  dcl 1 extended_test_my_fixed_type based,
        3 result                    fixed dec(31,0) init(0);
  dcl 1 extended_test_mylonglong_type based,
        3 result                    fixed bin(63) init(0);
  dcl 1 extended_test_myulonglong_type based,
        3 result                    fixed bin(63) init(0);
```
CHAPTER 7 | Orbix IDL Compiler

-L Argument

Overview

The -L argument gives you control over the generation of typedefs and typecodes. Its primary function is to reduce the amount of code generated to keep within the 10,000 line limit of the PL/I for MVS & VM compiler, but it also acts as a legacy flag for pre-Orbix 6.2 typedef generation. Qualifying the -L argument with a sub-option of c also allows you to generate z/OS UNIX System Services filenames in all uppercase.

Qualifying the -L argument

The -L argument must be qualified by c, i, s, u or su (that is, s and u combined). These options work as follows:

- c
  Generate filenames in all uppercase instead of all lowercase. This is relevant to z/OS UNIX System Services only.

- i
  Generate inherited typedefs (for pre-Orbix 6.2 compatibility). For example, by default, if interface B inherits A, and A contains an operation Fred, a typedef is generated for A/Fred only, because both B/Fred and A/Fred have the same signature.
  If -Li is specified, typedefs for both A/Fred and B/Fred are generated.

- s
  Generate only typecodes relating to sequences. Typedefs are only used by anys, sequences and CORBA::TypeCodes. If an application does not contain anys or CORBA::TypeCodes, this option can greatly reduce the number of typecodes generated.

- u
  Generate only typedefs that are referenced in the IDL. For example, if an IDL file contains a typedef called seq_short and seq_long, but only seq_short is used (for example, an operation signature), only the seq_short typedef is generated into the idlfilenameT include member.

- su
  This is a combination of the s and u options.
Specifying the -L argument

If you are running the Orbix IDL compiler in batch, the -L option is specified as follows:

```bash
// IDLPARM='-pli:-Lx',
```

If you are running the Orbix IDL compiler on z/OS UNIX System Services, the -L option is specified as follows:

```bash
idl -pli:-Lx
```

In the preceding two examples, x represents the sub-option i, s, u or su.

Specifying the -Lc argument

If you want to generate z/OS UNIX System Services filenames in all uppercase, you can specify the -Lc argument. For example:

```bash
idl -pli:Lc fred.idl
```

Based on the preceding command, the following files are then generated by the Orbix IDL compiler on z/OS UNIX System Services (assuming that the default extensions of .inc and .pli are used):

- FREDI.inc
- FREDI.pli
- FREDL.inc
- FREDM.inc
- FREDT.inc
- FREDV.pli
- FREDX.inc
-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, \textit{IDLmodule\_name(s)}\_IDLinterface\_name\_IDLvariable\_name). You can use the \textit{-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:

```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 25 by default for the specified interfaces:

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Generated Data Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SavingsBank</td>
<td>Banks_IrishBanks_SavingsBank</td>
</tr>
<tr>
<td>NationalBank</td>
<td>Banks_IrishBanks_NationalBank</td>
</tr>
<tr>
<td>DepositBank</td>
<td>Banks_IrishBanks_DepositBank</td>
</tr>
</tbody>
</table>

By using the \textit{-M} argument, you can replace the fully scoped names shown in Table 25 with alternative data names of your choosing.
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 \(\text{IDLMAP}\) 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 \(\text{orbixhlq}\) represents the high-level qualifier for your Orbix Mainframe installation):

```plaintext
//IDLMAP DD DISP=SHR,DSN=\text{orbixhlq}.DEMOS.PLI.MAP
```

You can define a DD card for \(\text{IDLMAP}\) 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.

The steps to generate alternative data name mappings with the \(-M\) argument are:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Run the Orbix IDL compiler with the (-M\text{create}) argument, to generate the mapping member, complete with the fully qualified names and their alternative mappings.</td>
</tr>
<tr>
<td>2</td>
<td>Edit (if necessary) the generated mapping member, to change the alternative name mappings to the names you want to use.</td>
</tr>
<tr>
<td>3</td>
<td>Run the Orbix IDL compiler with the (-M\text{process}) argument, to generate PL/I include members with the alternative data names.</td>
</tr>
</tbody>
</table>

First, you must run the Orbix IDL compiler with the \(-M\text{create}\) 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 \(\text{BANK}\) is the name of the mapping member you want to create):

```plaintext
IDLPARM='\(-\text{pli:-McreateXBANK}\)',
```
If you are running the Orbix IDL compiler in z/OS 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:-Mcreate\text{x}bank.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 z/OS 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 z/OS 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 \(/	ext{home/tcm/pli/map}\) directory:

```
-pli:-Dm/home/tcm/pli/map:-Mcreate\text{x}bank.map myfile.idl
```

See “\(-D\) Argument” on page 311 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 IDLinterface\(\text{name}_{(s)}\)IDLvariable\(\text{name}_{(s)}\)). The scope operator, /, is replaced with an underscore, _.

2 Map fully scoped IDL names to fully scoped PL/I names (that is, to IDL\(\text{modulename}\)IDLinterface\(\text{name}_{(s)}\)IDLvariable\(\text{name}_{(s)}\)). 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:

   ```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 26.

### Table 26: Example of Level-0-Scoped Generated Data Names

<table>
<thead>
<tr>
<th>Fully Scoped IDL Names</th>
<th>Generated Alternative Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banks</td>
<td>Banks</td>
</tr>
<tr>
<td>Banks/IrishBanks</td>
<td>IrishBanks</td>
</tr>
<tr>
<td>Banks/IrishBanks/SavingsBank</td>
<td>SavingsBank</td>
</tr>
<tr>
<td>Banks/IrishBanks/SavingsBank/accountbal</td>
<td>accountbal</td>
</tr>
<tr>
<td>Banks/IrishBanks/NationalBank</td>
<td>NationalBank</td>
</tr>
<tr>
<td>Banks/IrishBanks/NationalBank/deposit</td>
<td>deposit</td>
</tr>
</tbody>
</table>

Alternatively, based on the preceding IDL example, a `-Mcreate1BANK` command produces the `BANK` mapping member contents shown in Table 27.

### Table 27: Example of Level-1-Scoped Generated Data Names

<table>
<thead>
<tr>
<th>Fully Scoped IDL Names</th>
<th>Generated Alternative Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banks</td>
<td>Banks</td>
</tr>
<tr>
<td>Banks/IrishBanks</td>
<td>IrishBanks</td>
</tr>
</tbody>
</table>
Alternatively, based on the preceding IDL example, a `-Mcreate2BANK` command produces the BANK mapping member contents shown in Table 28.

**Table 27: Example of Level-1-Scoped Generated Data Names**

<table>
<thead>
<tr>
<th>Fully Scoped IDL Names</th>
<th>Generated Alternative Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banks/IrishBanks/SavingsBank</td>
<td>SavingsBank</td>
</tr>
<tr>
<td>Banks/IrishBanks/SavingsBank/accountbal</td>
<td>SavingsBank_accountbal</td>
</tr>
<tr>
<td>Banks/IrishBanks/NationalBank</td>
<td>NationalBank</td>
</tr>
<tr>
<td>Banks/IrishBanks/NationalBank/deposit</td>
<td>NationalBank_deposit</td>
</tr>
</tbody>
</table>

**Table 28: Example of Level-2-Scoped Generated Data Names**

<table>
<thead>
<tr>
<th>Fully Scoped IDL Names</th>
<th>Generated Alternative Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banks</td>
<td>Banks</td>
</tr>
<tr>
<td>Banks/IrishBanks</td>
<td>Banks_IrishBanks</td>
</tr>
<tr>
<td>Banks/IrishBanks/SavingsBank</td>
<td>Banks_IrishBanks_SavingsBank</td>
</tr>
<tr>
<td>Banks/IrishBanks/SavingsBank/accountbal</td>
<td>Banks_IrishBanks_SavingsBank_accountbal</td>
</tr>
<tr>
<td>Banks/IrishBanks/NationalBank</td>
<td>Banks_IrishBanks_NationalBank</td>
</tr>
<tr>
<td>Banks/IrishBanks/NationalBank/deposit</td>
<td>Banks_IrishBanks_NationalBank_deposit</td>
</tr>
</tbody>
</table>

**Note:** If two or more mapped names resolve to the same name, the Orbix IDL compiler completes with a return code of 4 and outputs a warning message similar to the following:

```
idl: "dd:IDLINC(MYINTF)", line 40: Warning: name mapping clash, my_intf/ping clashes with other_intf/ping. Both map to ping
```

It is the programmer’s responsibility to ensure that the mapping file is updated to ensure unique mapped names.
### 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 29: Example of Modified Mapping Names**

<table>
<thead>
<tr>
<th>Fully Scoped IDL Names</th>
<th>Modified Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banks/IrishBanks</td>
<td>IrishBanks</td>
</tr>
<tr>
<td>Banks/IrishBanks/SavingsBank</td>
<td>MyBank</td>
</tr>
<tr>
<td>Banks/IrishBanks/NationalBank</td>
<td>MyOtherBank</td>
</tr>
<tr>
<td>Banks/IrishBanks/SavingsBank/accountbal</td>
<td>Myaccountbalance</td>
</tr>
</tbody>
</table>

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 z/OS 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:-Mprocess bank.map
```

**Note:** If you are running the Orbix IDL compiler in z/OS 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 z/OS, 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 z/OS, 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:

```idl
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
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:

```plaintext
// SOURCE=TEST
// ...
// IDLPARM='-pli:-OSIMPLE'
```

If you are running the Orbix IDL compiler in z/OS 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` 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:

```plaintext
//   SOURCE=SIMPLE,
//   ...
//   IDLPARM='-pli:-S'
```

If you are running the Orbix IDL compiler in z/OS UNIX System Services, the following command, for example, creates a server implementation file called `SIMPLEI`, based on the `simple.idl` IDL file:

```plaintext
-pli:-S simple.idl
```

**Note:** In the case of z/OS UNIX System Services, if you use the `PLModuleExtension` 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 326 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, IMSG, 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 -TIMSx 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.

IMSx
Specifying -TIMSx generates IMS server mainline code.
Specifying -TIMSx with -S generates IMS server implementation code.
Specifying -TIMSx 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 -TIMSx with -V, it prevents the generation of IMS server mainline code.

IMSGx
This is similar to the IMSx option but does not generate the io_pcb_ptr and alt_pcb_ptr parameters. This option is provided to aid migration from Orbix 2.3-based IMS servers, which did not have these two parameter names.
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:

```plaintext
//   SOURCE=TEST,  
//   ... 
//   IDLPARM='-pli:-S:-TNATIVE',
```

If you are running the Orbix IDL compiler in z/OS 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:

```plaintext
-pli:-S:-TNATIVE test.idl
```

**Note:** Specifying -TNATIVE is the same as not specifying -T at all.

See “Developing the Server” on page 29 for an example of batch PL/I server mainline and implementation members.

Specifying the -TIMSx 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:

```plaintext
//   SOURCE=TEST,  
//   ... 
//   IDLPARM='-pli:-S:-TIMS4',
```

**Note:** Specifying -TIMSx generates IMS server mainline code. Specifying -TIMSx with -S generates IMS server implementation code.

**Note:** If you specify -TIMS 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.
If you are running the Orbix IDL compiler in z/OS 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 71 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 z/OS 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 143 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 \texttt{idlenameV} 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 \texttt{SIMPLE} IDL member, and prevents creation of a server mainline source code member called \texttt{SIMPLEV}:

```java
//   SOURCE=SIMPLE,
//   ...
//   IDLPARM='-pli:-V'
```

If you are running the Orbix IDL compiler in z/OS UNIX System Services, the following command, for example, only generates include files, based on the \texttt{simple.idl} IDL file, and prevents creation of a server mainline source code file called \texttt{SIMPLEV}:

```bash
-pli:-V simple.idl
```

**Note:** In the case of z/OS UNIX System Services, if you use the \texttt{PLModuleExtension} 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 326 for more details.
CHAPTER 7 | Orbix IDL Compiler

-W Argument

Overview
The -W argument allows you to specify whether you wish to have put skip or display messages generated in the idfilenameD include member. The default is put skip.

Qualifying the -W argument
The -W argument must be qualified by d or p. These options work as follows:

- **d**: Specifying -Wd generates display messages into the idfilenameD include member as follows:
  
  display('Error! No such operation:');
  display(operation);

- **p**: Specifying -Wp generates put skip messages into the idfilenameD include member as follows:
  
  put skip list('Error! No such operation:');
  put skip list(operation);

Specifying the -W argument
If you are running the Orbix IDL compiler in batch, the -W argument is specified as follows:

```
// IDLPARM='-pli:-Wx'
```

If you are running the Orbix IDL compiler on z/OS UNIX System Services, the -W argument is specified as follows:

```
-pli:-Wx
```

In the preceding two examples, x represents the sub-option d or p.
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:

<table>
<thead>
<tr>
<th>Configuration Variables</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL/I Configuration Variables</td>
<td>332</td>
</tr>
<tr>
<td>Adapter Mapping Member Configuration Variables</td>
<td>338</td>
</tr>
<tr>
<td>Providing Arguments to the IDL Compiler</td>
<td>341</td>
</tr>
</tbody>
</table>
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 z/OS 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 z/OS UNIX System Services.

Mandatory settings

If 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 30 provides an overview and description of the available configuration variables.

### Table 30: Summary of PL/I Configuration Variables (Sheet 1 of 3)

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsDefault</td>
<td>Indicates whether PL/I is the language that the Orbix IDL compiler generates by default from IDL. If this is set to <code>YES</code>, you do not need to specify the <code>-pli</code> switch when running the compiler.</td>
<td></td>
</tr>
<tr>
<td>PresetOptions</td>
<td>The PL/I plug-in arguments that are passed by default as parameters to the Orbix IDL compiler. Any arguments specified here do not need to be specified in the JCL or on the command line when running the Orbix IDL compiler.</td>
<td></td>
</tr>
<tr>
<td>PLIModuleExtension</td>
<td>Extension for the server source code filenames on z/OS UNIX System Services or Windows NT. <strong>Note:</strong> This is left blank by default, and you can set it to any value you want. The recommended setting is <code>.pli</code>.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 30: Summary of PL/I Configuration Variables (Sheet 2 of 3)

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLIIncludeExtensiona</td>
<td>Extension for PL/I include filenames on z/OS UNIX System Services or Windows NT.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This is left blank by default, and you can set it to any value you want. The recommended setting is .inc.</td>
<td></td>
</tr>
<tr>
<td>MainIncludeSuffix</td>
<td>Suffix for the main include member name.</td>
<td>M</td>
</tr>
<tr>
<td>TypedefIncludeSuffix</td>
<td>Suffix for the typedef include member name.</td>
<td>T</td>
</tr>
<tr>
<td>RuntimeIncludeSuffix</td>
<td>Suffix for the runtime include member name.</td>
<td>X</td>
</tr>
<tr>
<td>SelectIncludeSuffix</td>
<td>Suffix for the select include member name.</td>
<td>D</td>
</tr>
<tr>
<td>ServerMainModuleSuffix</td>
<td>Suffix for the server mainline source code member name.</td>
<td>V</td>
</tr>
<tr>
<td>ServerImplModuleSuffix</td>
<td>Suffix for the server implementation source code member name.</td>
<td>I</td>
</tr>
<tr>
<td>MaxFixedDigits</td>
<td>Maximum precision for the FIXED DECIMAL type.</td>
<td>15</td>
</tr>
<tr>
<td>NotSymbol</td>
<td>Symbol for the NOT operator.</td>
<td>¬</td>
</tr>
<tr>
<td>OrSymbol</td>
<td>Symbol for the OR operator.</td>
<td></td>
</tr>
</tbody>
</table>
Table 30: Summary of PL/I Configuration Variables  (Sheet 2 of 3)

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLIIncludeExtension</td>
<td>Extension for PL/I include filenames on z/OS UNIX System Services or Windows NT.</td>
<td></td>
</tr>
<tr>
<td>Note:</td>
<td>This is left blank by default, and you can set it to any value you want. The recommended setting is .inc.</td>
<td></td>
</tr>
<tr>
<td>MainIncludeSuffix</td>
<td>Suffix for the main include member name.</td>
<td>M</td>
</tr>
<tr>
<td>TypedefIncludeSuffix</td>
<td>Suffix for the typedef include member name.</td>
<td>T</td>
</tr>
<tr>
<td>RuntimeIncludeSuffix</td>
<td>Suffix for the runtime include member name.</td>
<td>X</td>
</tr>
<tr>
<td>SelectIncludeSuffix</td>
<td>Suffix for the select include member name.</td>
<td>D</td>
</tr>
<tr>
<td>ServerMainModuleSuffix</td>
<td>Suffix for the server mainline source code member name.</td>
<td>V</td>
</tr>
<tr>
<td>ServerImplModuleSuffix</td>
<td>Suffix for the server implementation source code member name.</td>
<td>I</td>
</tr>
<tr>
<td>MaxFixedDigits</td>
<td>Maximum precision for the FIXED DECIMAL type.</td>
<td>15</td>
</tr>
<tr>
<td>NotSymbol</td>
<td>Symbol for the NOT operator.</td>
<td>¬</td>
</tr>
<tr>
<td>OrSymbol</td>
<td>Symbol for the OR operator.</td>
<td>b</td>
</tr>
</tbody>
</table>
Table 30: Summary of PL/I Configuration Variables (Sheet 3 of 3)

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllCapsFilenames</td>
<td>Indicates whether to generate filenames on z/OS UNIX System Services in all uppercase, if this is set to YES. This is equivalent to specifying the -lc option with the Orbix IDL compiler.</td>
<td>NO</td>
</tr>
<tr>
<td>EnterpriseEnabled</td>
<td>Enables Enterprise PL/I options supported by the PL/I generator, if set to YES. If this is set to YES, it overrides the MaxFixedDigits setting. This is equivalent to specifying the -E option with the Orbix IDL compiler.</td>
<td>NO</td>
</tr>
<tr>
<td>MessageStatement</td>
<td>Indicates whether to generate output messages as display (if set to D) or put skip statements (if set to P or not set at all). This is equivalent to specifying the -wd or -wp option with the Orbix IDL compiler.</td>
<td>P</td>
</tr>
</tbody>
</table>

a. This is ignored on native z/OS.
b. PL/I uses a double OR symbol (that is, ||) as a string concatenation symbol.

The last nine variables shown in Table 30 on page 333 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:

```plaintext
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 = ""
};
```
Orbix IDL Compiler Configuration

**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 31 provides an overview and description of the available configuration variables.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsDefault</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>PresetOptions</td>
<td>The PL/I arguments that are passed by default as parameters to the Orbix IDL compiler for the purposes of generating adapter mapping members. Any arguments specified here do not need to be specified in the JCL or on the command line when running the Orbix IDL compiler.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 31: Adapter Mapping Member Configuration Variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFAMappingExtension</td>
<td>Extension for the adapter mapping filename on z/OS UNIX System Services or Windows NT.</td>
<td>map</td>
</tr>
<tr>
<td>MFAMappingSuffix</td>
<td>Suffix for the adapter mapping member name. If you do not specify a value for this, it is generated with an <code>A</code> suffix by default.</td>
<td>A</td>
</tr>
<tr>
<td>TypeInfoFileExtension</td>
<td>Extension for the type information filename on z/OS UNIX System Services or Windows NT.</td>
<td>inf</td>
</tr>
<tr>
<td>TypeInfoFileSuffix</td>
<td>Suffix for the type information member name. If you do not specify a value for this, it is generated with a <code>B</code> suffix by default.</td>
<td>B</td>
</tr>
</tbody>
</table>

a. This is ignored on native z/OS.
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 332 for an example of the Pli configuration scope. See “Adapter Mapping Member Configuration Variables” on page 338 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 PresetOptions variable in the main Orbix IDL compiler configuration data set. If a value is specified in the PresetOptions 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=SOURCE01,
//         IDL=&ORBIX..DEMOS.IDL,
//         COPYLIB=&ORBIX..DEMOS.PLINCL,
//         IMPL=&ORBIX..DEMOS.PLISRC,
//         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 332 for an example of the Pli configuration scope. See “Adapter Mapping Member Configuration Variables” on page 338 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=SOURCE01,
//         IDL=&ORBIX..DEMOS.IDL,
//         COPYLIB=&ORBIX..DEMOS.CICS.PLINCL,
//         IMPL=&ORBIX..DEMOS.CICS.PLISRC,
//         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
//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:

<table>
<thead>
<tr>
<th>Operation Parameters</th>
<th>page 346</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Management Routines</td>
<td>page 371</td>
</tr>
</tbody>
</table>

Note: See “API Reference” on page 377 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:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded Sequences and Memory Management</td>
<td>347</td>
</tr>
<tr>
<td>Unbounded Sequences and Memory Management</td>
<td>351</td>
</tr>
<tr>
<td>Unbounded Strings and Memory Management</td>
<td>356</td>
</tr>
<tr>
<td>The any Type and Memory Management</td>
<td>364</td>
</tr>
<tr>
<td>User Exceptions and Memory Management</td>
<td>369</td>
</tr>
</tbody>
</table>
Bounded Sequences and Memory Management

Overview for IN parameters

Table 32 provides a detailed outline of how memory is handled for bounded sequences that are used as IN parameters.

Table 32: Memory Handling for IN Bounded Sequences

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Server Application</th>
</tr>
</thead>
</table>
| 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 32:

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 33 provides a detailed outline of how memory is handled for bounded sequences that are used as inout parameters.

### Table 33: Memory Handling for INOUT Bounded Sequences

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Server Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. SEQINIT</td>
<td>4. PODGET—(receive, allocate)</td>
</tr>
<tr>
<td>2. write</td>
<td>5. read</td>
</tr>
<tr>
<td>3. PODEXEC—(send)</td>
<td>6. SEQFREE</td>
</tr>
<tr>
<td></td>
<td>7. SEQINIT</td>
</tr>
<tr>
<td>10. (free, receive, allocate)</td>
<td>8. write</td>
</tr>
<tr>
<td>11. read</td>
<td>9. PODPUT—(send, free)</td>
</tr>
<tr>
<td>12. SEQFREE</td>
<td></td>
</tr>
</tbody>
</table>

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 33:

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 34 provides a detailed outline of how memory is handled for bounded sequences that are used as `out` or `return` parameters.

**Table 34:** Memory Handling for OUT and Return Bounded Sequences

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Server Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <code>PODEXEC</code>—(send)</td>
<td>2. <code>PODGET</code>—(receive)</td>
</tr>
<tr>
<td></td>
<td>3. <code>SEQINIT</code></td>
</tr>
<tr>
<td></td>
<td>4. write</td>
</tr>
<tr>
<td>6. (receive, allocate)</td>
<td>5. <code>PODPUT</code>—(send, free)</td>
</tr>
<tr>
<td>7. read</td>
<td></td>
</tr>
<tr>
<td>8. <code>SEQFREE</code></td>
<td></td>
</tr>
</tbody>
</table>
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 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 `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 35 provides a detailed outline of how memory is handled for unbounded sequences that are used as in parameters.

Table 35: Memory Handling for IN Unbounded Sequences

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Server Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. SEQALLOC</td>
<td>4. PODGET—(receive, allocate)</td>
</tr>
<tr>
<td>2. SEQSET</td>
<td>5. SEQGET</td>
</tr>
<tr>
<td>3. PODEXEC—(send)</td>
<td>6. PODPUT—(free)</td>
</tr>
<tr>
<td>7. SEQFREE</td>
<td></td>
</tr>
</tbody>
</table>

a. SEQSET performs a deep copy from the element buffer into the sequence. This means that if an element buffer contains dynamic data (for example, a string or a sequence), the element buffer should be freed after calling SEQSET, to prevent memory leaks. Memory should be handled as follows for an unbounded sequence of strings, to prevent a leak:
1. Call STRSET to allocate an element in the element buffer.
2. Call SEQSET to copy the element into the sequence.
3. Call STRFREE to free the element buffer.

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 35:

1. The client calls SEQALLOC 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 36 provides a detailed outline of how memory is handled for unbounded sequences that are used as `inout` parameters.

**Table 36: Memory Handling for INOUT Unbounded Sequences**

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Server Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. SEQALOC</td>
<td>4. PODGET—(receive, allocate)</td>
</tr>
<tr>
<td>2. SEQSET a</td>
<td>5. SEQGET</td>
</tr>
<tr>
<td>3. PODEXEC—(send)</td>
<td>6. SEQFREE</td>
</tr>
<tr>
<td>7. SEQALOC</td>
<td>8. SEQGET</td>
</tr>
<tr>
<td>8. SEQSET</td>
<td>9. PODPUT—(send, free)</td>
</tr>
<tr>
<td>10. (free, receive, allocate)</td>
<td></td>
</tr>
<tr>
<td>11. SEQGET</td>
<td></td>
</tr>
<tr>
<td>12. SEQFREE</td>
<td></td>
</tr>
</tbody>
</table>

a. SEQSET performs a deep copy from the element buffer into the sequence. This means that if an element buffer contains dynamic data (for example, a string or a sequence), the element buffer should be freed after calling SEQSET, to prevent memory leaks. Memory should be handled as follows for an unbounded sequence of strings, to prevent a leak:

1. Call STRSET to allocate an element in the element buffer.
2. Call SEQSET to copy the element into the sequence.
3. Call STRFREE to free the element buffer.

### 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 36:

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 SEQALLOC to initialize the replacement out sequence and allocate memory for both the sequence information block and the sequence data.

8. The server calls SEQGET 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 SEQALLOC 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 SEQALLOC 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 37 provides a detailed outline of how memory is handled for unbounded sequences that are used as out or return parameters.

**Table 37: Memory Handling for OUT and Return Unbounded Sequences**

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Server Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. PODEXEC—(send)</td>
<td>2. PODGET—(receive)</td>
</tr>
<tr>
<td>3. SEQALOC</td>
<td>4. SEQSETa</td>
</tr>
<tr>
<td>4. SEQSET</td>
<td>5. PODPUT—(send, free)</td>
</tr>
<tr>
<td>6. (receive, allocate)</td>
<td></td>
</tr>
<tr>
<td>7. SEQGET</td>
<td></td>
</tr>
<tr>
<td>8. SEQFREE</td>
<td></td>
</tr>
</tbody>
</table>

a. SEQSET performs a deep copy from the element buffer into the sequence. This means that if an element buffer contains dynamic data (for example, a string or a sequence), the element buffer should be freed after calling SEQSET, to prevent memory leaks. Memory should be handled as follows for an unbounded sequence of strings, to prevent a leak:
1. Call STRSET to allocate an element in the element buffer.
2. Call SEQSET to copy the element into the sequence.
3. Call STRFREE to free the element buffer.

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 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 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 38 provides a detailed outline of how memory is handled for unbounded strings that are used as in parameters.

Table 38: Memory Handling for IN Unbounded Strings

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Server Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. STRSET</td>
<td>3. PODGET—(receive, allocate)</td>
</tr>
<tr>
<td>2. PODEXEC—(send)</td>
<td>4. STRGET</td>
</tr>
<tr>
<td>6. STRFREE</td>
<td>5. PODPUT—(free)</td>
</tr>
</tbody>
</table>

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 38:

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 39 provides a detailed outline of how memory is handled for unbounded strings that are used as inout parameters.

Table 39: Memory Handling for INOUT Unbounded Strings

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Server Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. STRSET</td>
<td>3. PODGET—(receive, allocate)</td>
</tr>
<tr>
<td>2. PODEXEC—(send)</td>
<td>4. STRGET</td>
</tr>
<tr>
<td>8. (free, receive, allocate)</td>
<td>5. STRFREE</td>
</tr>
<tr>
<td>9. STRGET</td>
<td>6. STRSET</td>
</tr>
<tr>
<td>10. STRFREE</td>
<td>7. PODPUT—(send, free)</td>
</tr>
</tbody>
</table>

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 39:

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 40 provides a detailed outline of how memory is handled for unbounded strings that are used as out or return parameters.

**Table 40: Memory Handling for OUT and Return Unbounded Strings**

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Server Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. PODEXEC—(send)</td>
<td>2. PODGET—(receive)</td>
</tr>
<tr>
<td>5. (receive, allocate)</td>
<td>3. STRSET</td>
</tr>
<tr>
<td>6. STRGET</td>
<td>4. PODPUT—(send, free)</td>
</tr>
</tbody>
</table>

**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 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 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 41 provides a detailed outline of how memory is handled for object references that are used as in parameters.

Table 41: Memory Handling for IN Object References

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Server Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Attain object reference</td>
<td>3. PODGET—(receive)</td>
</tr>
<tr>
<td>2. PODEXEC—(send)</td>
<td>4. read</td>
</tr>
<tr>
<td>6. OBJREL</td>
<td>5. PODPUT</td>
</tr>
</tbody>
</table>

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 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.
Operation Parameters

Overview for INOUT parameters

Table 42 provides a detailed outline of how memory is handled for object references that are used as in parameters.

Table 42: Memory Handling for INOUT Object References

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Server Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Attain object reference</td>
<td>3. PODGET—(receive)</td>
</tr>
<tr>
<td>2. PODEXEC—(send)</td>
<td>4. read</td>
</tr>
<tr>
<td>5. OBJREL</td>
<td>6. Attain object reference</td>
</tr>
<tr>
<td>7. OBJDUPL</td>
<td>8. PODPUT—(send)</td>
</tr>
<tr>
<td>9. (receive)</td>
<td>10. read</td>
</tr>
<tr>
<td>11. OBJREL</td>
<td>12. OBJREL</td>
</tr>
</tbody>
</table>

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 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.

### Table 43: Memory Handling for OUT and Return Object References

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Server Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. PODEXEC—(send)</td>
<td>2. PODGET—(receive)</td>
</tr>
<tr>
<td>6. (receive)</td>
<td>3. Attain object reference</td>
</tr>
<tr>
<td>7. read</td>
<td>4. OBJDUPL</td>
</tr>
<tr>
<td>8. OBJREL</td>
<td>5. PODPUT—(send)</td>
</tr>
</tbody>
</table>

### 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 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 44 provides a detailed outline of how memory is handled for an any type that is used as an in parameter.

Table 44: Memory Handling for IN Any Types

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Server Application</th>
</tr>
</thead>
</table>
| 1. TYPESET  
2. ANYSET  
3. PODEXEC—(send) | 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 44:

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.
Table 45 provides a detailed outline of how memory is handled for an any type that is used as an inout parameter.

### Table 45: Memory Handling for INOUT Any Types

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Server Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. TYPESET</td>
<td>4. PODGET—(receive, allocate)</td>
</tr>
<tr>
<td>2. ANYSET</td>
<td>5. TYPEGET</td>
</tr>
<tr>
<td>3. PODEXEC—(send)</td>
<td>6. ANYGET</td>
</tr>
<tr>
<td></td>
<td>7. ANYFREE</td>
</tr>
<tr>
<td>8. TYPEGET</td>
<td>9. ANYSET</td>
</tr>
<tr>
<td>9. ANYSET</td>
<td>10. PODPUT—(send, free)</td>
</tr>
<tr>
<td>11. (free, receive, allocate)</td>
<td></td>
</tr>
<tr>
<td>12. TYPEGET</td>
<td></td>
</tr>
<tr>
<td>13. ANYGET</td>
<td></td>
</tr>
<tr>
<td>14. ANYFREE</td>
<td></td>
</tr>
</tbody>
</table>

The memory handling rules for an object reference used as an inout parameter can be summarized as follows, based on Table 45:

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.
Table 46 provides a detailed outline of how memory is handled for an any type that is used as an inout parameter.

### Table 46: Memory Handling for OUT and Return Any Types

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Server Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. PODEXEC—(send)</td>
<td>2. PODGET—(receive)</td>
</tr>
<tr>
<td>6. (receive, allocate)</td>
<td>3. TYPESET</td>
</tr>
<tr>
<td>7. TYPEGET</td>
<td>4. ANYSET</td>
</tr>
<tr>
<td>8. ANYGET</td>
<td>5. PODPUT—(send, free)</td>
</tr>
<tr>
<td>9. ANYFREE</td>
<td></td>
</tr>
</tbody>
</table>

The memory handling rules for an object reference used as an out or return parameter can be summarized as follows, based on Table 46:

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 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 \texttt{ANYFREE}, which causes the client-side PL/I runtime to free the memory allocated for the \texttt{any} in point 6 via the call to \texttt{PODEXEC} in point 1.
User Exceptions and Memory Management

Overview

Table 47 provides a detailed outline of how memory is handled for user exceptions.

Table 47: Memory Handling for User Exceptions

<table>
<thead>
<tr>
<th>Client Application</th>
<th>Server Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. PODEXEC—(send)</td>
<td>2. PODGET—(receive, allocate)</td>
</tr>
<tr>
<td>3. write</td>
<td>4. PODERR</td>
</tr>
<tr>
<td>5. (free)</td>
<td></td>
</tr>
<tr>
<td>6. Free</td>
<td></td>
</tr>
</tbody>
</table>

Summary of rules

The memory handling rules for raised user exceptions can be summarized as follows, based on Table 47:

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:

```pli
/* 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 WSTRFREE to subsequently free this memory. For example:

```pli
/* allocation */
dcl my_corba_wstring    ptr;
call wstrset(my_pli_graphic, my_corba_wstring,
             my_pli_graphic_length);

/* deletion */
call wstrfree(my_corba_wstring);
```
Typecodes

As described in “IDL-to-PL/I Mapping” on page 235, 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:

```pli
/* 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:

```pli
/* 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 **SEQALLOC**. 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 **SEQALLOC** 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. If an attempt is made to add an element beyond the maximum size of the sequence, **SEQSET** automatically resizes the sequence for you by adding
1024 elements to the sequence maximum. If the sequence size grows larger than 8K, the resize amount is calculated as follows:

sequence maximum + (1/8 * current sequence maximum).

**Note:** Additional overhead is incurred by your application each time a resize occurs. This is because an allocation, a copy, and a free occur each time. The larger your sequence, the larger your overhead. To avoid this overhead, ensure you specify the sequence maximum in your application.

Use `SEQFREE` to free memory allocated via `SEQALOC`. For example:

```c
/* allocation */
call seqalloc(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:

```c
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

This part contains the following chapters:

| API Reference       | page 377 |
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.

In this chapter

This chapter discusses the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Reference Summary</td>
<td>378</td>
</tr>
<tr>
<td>API Reference Details</td>
<td>384</td>
</tr>
<tr>
<td>Deprecated and Removed APIs</td>
<td>506</td>
</tr>
</tbody>
</table>
# API Reference Summary

## Introduction

This section provides a summary of the available API functions, in alphabetic order. See “API Reference Details” on page 384 for more details of each function.

## Summary listing

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANYFREE(inout PTR any_pointer)</td>
<td>Frees memory allocated to an any.</td>
</tr>
<tr>
<td>ANYGET(in PTR any_pointer, out PTR any_data_buffer)</td>
<td>Extracts data out of an any.</td>
</tr>
<tr>
<td>ANYSET(inout PTR any_pointer, in PTR any_data_buffer)</td>
<td>Inserts data into an any.</td>
</tr>
<tr>
<td>MEMALLOC(out PTR memory_pointer, in FIXED BIN(31) memory_size)</td>
<td>Allocates memory at runtime from the program heap.</td>
</tr>
<tr>
<td>MEMDBG(in PTR memory_pointer, in FIXED BIN(15) memory_dump_size, in CHAR(*) text_string, in FIXED BIN(15) text_string_length)</td>
<td>Output a formatted memory dump for the specified block of memory.</td>
</tr>
<tr>
<td>MEMFREE(in PTR memory_pointer)</td>
<td>Frees the memory allocated at the address passed in.</td>
</tr>
<tr>
<td>OBJGTID(in PTR object_reference, out CHAR(*) object_id, in FIXED BIN(31) object_id_length)</td>
<td>Retrieves the object ID from an object reference.</td>
</tr>
</tbody>
</table>
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.

PODTXNB
// Indicates the beginning of a two-phase commit transaction.

PODTXNE
// Indicates the end of a two-phase commit transaction.

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

SEQSET(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

SEQREL(in PTR sequence_control_data,
   in CHAR(*) typecode_key,
   in FIXED BIN(31) typecode_key_length)
// Frees the memory allocated to an unbounded sequence and its contents

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
CHAPTER 9 | API Reference

`STRLEN` (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.

`WSTFREE` (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
**API Reference Summary**

**WSTRLEN**

```plaintext
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**

```plaintext
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**

```plaintext
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**

```plaintext
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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANYFREE</td>
<td>387</td>
</tr>
<tr>
<td>ANYGET</td>
<td>389</td>
</tr>
<tr>
<td>ANYSET</td>
<td>391</td>
</tr>
<tr>
<td>MEMALOC</td>
<td>393</td>
</tr>
<tr>
<td>MEMDEBUG</td>
<td>394</td>
</tr>
<tr>
<td>MEMFREE</td>
<td>396</td>
</tr>
<tr>
<td>OBJDUPL</td>
<td>397</td>
</tr>
<tr>
<td>OBJGTID</td>
<td>399</td>
</tr>
<tr>
<td>OBJNEW</td>
<td>401</td>
</tr>
<tr>
<td>OBJREL</td>
<td>403</td>
</tr>
<tr>
<td>OBJRIR</td>
<td>405</td>
</tr>
<tr>
<td>OBJ2STR</td>
<td>407</td>
</tr>
<tr>
<td>ORBARGS</td>
<td>409</td>
</tr>
<tr>
<td>PODERR</td>
<td>413</td>
</tr>
<tr>
<td>PODEXEC</td>
<td>418</td>
</tr>
<tr>
<td>PODGET</td>
<td>421</td>
</tr>
<tr>
<td>PODINFO</td>
<td>424</td>
</tr>
<tr>
<td>PODPUT</td>
<td>426</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td>PODREG</td>
<td>429</td>
</tr>
<tr>
<td>PODREQ</td>
<td>431</td>
</tr>
<tr>
<td>PODRUN</td>
<td>434</td>
</tr>
<tr>
<td>PODSRVR</td>
<td>435</td>
</tr>
<tr>
<td>PODSTAT</td>
<td>437</td>
</tr>
<tr>
<td>PODTIME</td>
<td>440</td>
</tr>
<tr>
<td>PODTXNB</td>
<td>442</td>
</tr>
<tr>
<td>PODTXNE</td>
<td>443</td>
</tr>
<tr>
<td>PODVER</td>
<td>444</td>
</tr>
<tr>
<td>SEQALOC</td>
<td>445</td>
</tr>
<tr>
<td>SEQDUPL</td>
<td>448</td>
</tr>
<tr>
<td>SEQFREE</td>
<td>450</td>
</tr>
<tr>
<td>SEQGET</td>
<td>452</td>
</tr>
<tr>
<td>SEGINIT</td>
<td>455</td>
</tr>
<tr>
<td>SEQLEN</td>
<td>457</td>
</tr>
<tr>
<td>SQLSET</td>
<td>459</td>
</tr>
<tr>
<td>SEQMAX</td>
<td>462</td>
</tr>
<tr>
<td>SEQREL</td>
<td>465</td>
</tr>
<tr>
<td>SEQSET</td>
<td>467</td>
</tr>
<tr>
<td>STRCON</td>
<td>470</td>
</tr>
<tr>
<td>STRDUPL</td>
<td>472</td>
</tr>
<tr>
<td>STRFREE</td>
<td>473</td>
</tr>
<tr>
<td>STRGET</td>
<td>474</td>
</tr>
<tr>
<td>STRLENG</td>
<td>476</td>
</tr>
<tr>
<td>STRSET</td>
<td>478</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
</tr>
<tr>
<td>STRSETS</td>
<td>480</td>
</tr>
<tr>
<td>STR2OBJ</td>
<td>481</td>
</tr>
<tr>
<td>TYPEGET</td>
<td>486</td>
</tr>
<tr>
<td>TYPESET</td>
<td>489</td>
</tr>
<tr>
<td>WSTRCON</td>
<td>491</td>
</tr>
<tr>
<td>WSTRDUP</td>
<td>493</td>
</tr>
<tr>
<td>WSTRFRE</td>
<td>494</td>
</tr>
<tr>
<td>WSTRGET</td>
<td>496</td>
</tr>
<tr>
<td>WSTRELEN</td>
<td>498</td>
</tr>
<tr>
<td>WSTRSET</td>
<td>500</td>
</tr>
<tr>
<td>WSTRSTS</td>
<td>502</td>
</tr>
<tr>
<td>CHECK_ERRORS</td>
<td>503</td>
</tr>
</tbody>
</table>
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:

   ```idl
   interface test {
     attribute any myany;
   };
   ```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the idlmembernameT include member (where idlmembername represents the name of the IDL member that contains the IDL definitions):

   ```idl
   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 \textit{idlmembername} include member:

\begin{verbatim}
dcl l test_myany_attr aligned like test_myany_type;
\end{verbatim}

3. The following is an example of how to use \texttt{ANYFREE} in association with the preceding code:

\begin{verbatim}
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 */
call anyfree(test_myany_attr.result);
\end{verbatim}

\textbf{See also}

- “\texttt{ANYSET}” on page 391.
- “\texttt{TYPESET}” on page 489.
- “Memory Handling” on page 345.
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:

   ```idl
   interface test {
       attribute any myany;
   };
   ```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the idlmembername include member (where idlmembername represents the name of the IDL member that contains the IDL definitions):

   ```idl
   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 \texttt{idlmembername} include member:

\begin{verbatim}
dcl 1 test_myany_attr aligned like test_myany_type;

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);
\end{verbatim}

See also

"ANYSET" on page 391.
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:

   ```idl
   interface test {
     attribute any myany;
   };
   ```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the idlmembernameT include member (where idlmembername represents the name of the IDL member that contains the IDL definitions):

   ```idl
   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 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. */
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 389.
- “TYPESET” on page 489.
**MEMALOC**

**Synopsis**

```plaintext
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:

```plaintext
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 396.
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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory_pointer</td>
<td>This is an in parameter that contains a pointer to the allocated memory block.</td>
</tr>
<tr>
<td>memory_dump_size</td>
<td>This is an in parameter that specifies in bytes the amount of memory that is to be allocated for the memory dump.</td>
</tr>
<tr>
<td>text_string</td>
<td>This is an in parameter that contains the text string relating to the memory dump.</td>
</tr>
<tr>
<td>text_string_length</td>
<td>This is an in parameter that specifies the length of the text string.</td>
</tr>
</tbody>
</table>

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): 00000000 E2E3C1E3 C9E2E3C9 C3E20000 '..âstrln9..
00x3a598 (00040): A3998995 8700FFFF 40400400 '..&str1n9.. '
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 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:

```plaintext
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 393.
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:

```fortran
! Example of using OBJDUPL

dcl l get_an_object_args,
    3 result                       ptr init(sysnull());
dcl test_prq_object                ptr init(sysnull());
dcl my_object                      ptr init(sysnull());

! test_prq_object already set up from earlier processing
! call podexec(test_prq_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 403 and “Object References and Memory Management” on page 360.
API Reference Details

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:

```c

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 idlmembernameT 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):

```plaintext
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 PODSsrvr 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:

```plaintext

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 397 and “Object References and Memory Management” on page 360.
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 CORBA service 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 48 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 48: Summary of Common Services and Their PL/I Identifiers

<table>
<thead>
<tr>
<th>Service</th>
<th>PL/I Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>InterfaceRepository</td>
<td>IFR_SERVICE</td>
</tr>
<tr>
<td>NameService</td>
<td>NAMING_SERVICE</td>
</tr>
<tr>
<td>TradingService</td>
<td>TRADING_SERVICE</td>
</tr>
</tbody>
</table>

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.
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`):

```plaintext
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 \texttt{idlmembernameSV} member by the Orbix IDL compiler, with added comments for clarity:

\begin{verbatim}
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(iorrec_ptr,iorrec,iorrec_len);
if check_errors('strget') ^= completion_status_yes then return;
write file(IORFILE) FROM(iorrec);
close file(IORFILE);
\end{verbatim}

See also

“STR2OBJ” on page 481.
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.

The ORB identifier (specified via the -ORBid argument) is defined by the CORBA specification. It is intended to uniquely identify ORBs used within the same process in a multi-ORB application. The value specified for -ORBid is set on ORB initialization during the call to CORBA::ORB_init() in C++.

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 specific to Orbix CORBA ORBs and is used to select a configuration scope from within a configuration domain. The value specified for `-ORBname` is also set on ORB initialization, based on the following logic:

1. If a `-ORBname` value is passed as a parameter to `ORBARGS`, use that value.
2. Check for the existence of the environment variable `IT_ORB_NAME`, and use its value if set.
3. Use the `-ORBid` value.

---

### Example

The following is an example of client code at ORB setup time:

```plaintext
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;
```

...
CHAPTER 9 | API Reference

Exceptions

A CORBA::BAD_INV_ORDER::ADAPTER_ALREADY_INITIALIZED exception is raised if ORBARGS is called more than once in a client or server.

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.
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 345 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 418 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.
The example can be broken down as follows:

1. Consider the following IDL:

```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` include member (where `idlmembername` represents the name of the IDL member that contains the IDL definitions):

```idl
/*-----------------------------------------------*/
/* 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);
dcl 1 test_myop_args aligned like test_myop_type;

dcl 1 test_myop_type based,
   3 number  fixed bin(31)  init(0),
   3 result  fixed bin(31)  init(0);
```

The following operation structure declaration is also generated in the `idlmembername` include member:

```idl
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`:

```c
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:

```c
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_userexceptions);
   end;
else
   do;...
```
5. A test such as the following can be set up in the client code to check for a user exception:

```plaintext
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 418.
- 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.
The example can be broken down as follows:

1. Consider the following IDL:

```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 include member (where `idlmembername` represents the name of the IDL member that contains the IDL definitions):

```idl
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` include member:

```plaintext
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:

```plaintext
/* 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

- **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.
- **CORBA::BAD_PARAM::INVALID_DISCRIMINATOR_TYPECODE** exception is raised if the discriminator typecode is invalid when marshalling a union type.
- **CORBA::BAD_PARAM::UNKNOWN_OPERATION** exception is raised if the operation is not valid for the interface.
- **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` 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:

   ```plaintext
   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 `IDLMemberNameT` include member (where `IDLMemberName` represents the name of the IDL member that contains the IDL definitions):

```plaintext
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` include member is as follows:

```plaintext
dcl 1 foo_bar_args aligned like foo_bar_type;
```

4. A subset of the `IDLMemberNameD` include member is as follows, with comments added for clarity:

```plaintext
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 426.
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 idlmembernameV 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:

```c
/* 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 437.
### 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 413 for more details.

The Orbix IDL compiler generates the call for `PODPUT` in the `idlmembernameD` 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:

   ```plaintext
   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 `idlmembernameT` 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 `idlmembernameM` include member is as follows:

```
dcl 1 foo_bar_args aligned like foo_bar_type;
```

4. A subset of the `idlmembernameD` 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 421.
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 idlmembernameX include member, which the Orbix IDL compiler generates.

The Orbix IDL compiler generates an idlmembernameX include member for each IDL interface (where idlmembername represents the name of the IDL member that contains the IDL definitions). The idlmembernameX 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 idlmembernameX 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:

```pli
/* 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:

```plaintext
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.
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.

The example can be broken down as follows:

1. The following code is in the idlmembername server implementation module, generated by the Orbix IDL compiler (where idlmembername represents the name of the IDL member that contains the IDL definitions):

```pli
/* Entry point to enable the Orbix PL/I runtime to call */
/* out to the server implementation for when a request */
/* comes in. */
DISPTCH: ENTRY;

/* 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;
```

2. The following code is in the supplied DISPINIT include member that the server implementation includes:
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 474.
PODRUN

**Synopsis**

```
// 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 `idlmembernameV` 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:

```pli
/* 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:

```plaintext
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 idlmembernameV 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` (that is, server mainline) module:

```plaintext
%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.
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 503.
**PODTIME**

**Synopsis**

```plaintext
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. |
PODTXNB

**Synopsis**

PODTXNB

// Indicates the beginning of a two-phase commit transaction

**Usage**

Client-specific. Only supported for CICS and IMS clients.

**Description**

The PODTXNB function marks the beginning of two-phase commit processing. Any update calls to servers, made using PODEXEC after a call to PODTXNB, send data over a sync level 2 APPC conversation. This allows for committing or rolling back the updates made using PODEXEC.

**Parameters**

PODTXNB takes no parameters.

**Example**

The following is an example of how to call PODTXNB:

```plaintext
--
call podtxnb;
if check_errors('podtxnb') ^= completion_status_yes
  then return;
```
PODTXNE

Synopsis

PODTXNE

// Indicates the end of a two-phase commit transaction

Usage

Client-specific. Only supported for CICS and IMS clients.

Description

The PODTXNE function marks the end of two-phase commit processing. This function requests that the sync level 2 APPC conversation is deallocated. Very little processing should take place after this call. There should be no more calls to PODEXEC.

Parameters

PODTXNE takes no parameters.

Example

The following is an example of how to call PODTXNE:

```plaintext
...
call podtxne;
if check_errors('podtxne') ^= completion_status_yes
  then return;
```
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 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:

```pli
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 idlmembernameT 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 `idlmembernameT` module (where `idlmembername` represents the name of the IDL member that contains the IDL definitions):

   ```
   /* Extract from EXAMPLET */
   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 `idlmembernameM` module:

   ```
   /* Extract from EXAMPLEM */
   dcl 1 example_myseq_attr aligned like example_myseq_type;
   ```

3. The following example shows how the code generated in the `idlmembernameT` and `idlmembernameM` modules can then be used by the user’s implementation code in the `idlmembernameI` module:

   ```
   /* Extract from EXAMPLEI showing some of the user's */
   /* implementation. Allocate space for 20 elements */
   /* in the unbounded sequence myseq. */
   call seqalloc(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 450.
- “SEQSET” on page 467.
- “Memory Handling” on page 345.
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):

```plaintext
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);

/* seqalloc 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 450.
- “Memory Handling” on page 345.
** 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 SEQALLOC 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 `idlmembename` module (where `idlmembename` represents the name of the IDL member that contains the IDL definitions):

   ```
   /* Extract from EXAMPLE */
   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 idlmembernameM module:

```c
/* Extract from EXAMPLEM */
dcl 1 example_myseq_attr aligned like example_myseq_type;

3. The following example shows how the code generated in the idlmembernameI and idlmembernameM modules can be used by the user's implementation code in the idlmembernameI module:

```c
/* Extract from EXAMPLEI showing some of the user's implementation. Our unbounded sequence gets initialized */
/* with 25 elements' space */
call seqalloc(example_myseq_attr.result.result_seq,
             25,
             example_seqlong_tc,
             length(example_seqlong_tc));
if check_errors('seqalloc') ^= completion_status_yes then
  return;
/* Processing omitted */
```

```c
/* Finished working with the unbounded sequence, now free it */
call seqfree(example_myseq.result.result_seq);
```

See also

- “SEQALLOC” on page 445.
- “Memory Handling” on page 345.
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 sequence into the element buffer associated with this sequence (that is, into the sequence-data parameter).

Note: This copy is a shallow copy, so pointers to dynamic data areas should be handled with care.

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:

   ```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 \texttt{idlmembernameT} module (where \texttt{idlmembername} represents the name of the IDL member that contains the IDL definitions):

\begin{verbatim}
/* Extract from EXAMPLET */
dcl 1 example_myseq_type based, 3 result,
    5 result_seq ptr init(sysnull()),
    5 result_buf fixed bin (31) init(0);
\end{verbatim}

3. Based on the preceding IDL, the Orbix IDL compiler also generates the following code, in the \texttt{idlmembernameM} module:

\begin{verbatim}
/* Extract from EXAMPLEM */
dcl 1 example_myseq_attr aligned like example_myseq_type;
\end{verbatim}
The following example shows how the code generated in the `idlmembernameT` and `idlmembernameM` modules can then be used by the user’s implementation code in a client program:

```cpp
/* 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 467.
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 `idlmembernameT` 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 `idlmembernameM` 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 `idlmembernameT` and `idlmembernameM` modules can then be used by the user’s implementation code in the `idlmembernameI` 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 459.
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:

```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 module (where idlmembername represents the name of the IDL member that contains the IDL definitions):

```idl
/* Extract from EXAMPLE */
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` module:

```c
/* 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` and `idlmembername` modules can then be used by the user's implementation code in a client program:

```c
/* 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 462.
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:

```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* module (where *idlmembername* represents the name of the IDL member that contains the IDL definitions):

```idl
/* Extract from EXAMPLET */
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* module:

```idl
/* Extract from EXAMPLEM */
dcl 1 example_myseq_attr aligned like example_myseq_type;
```
3. The following example shows how the code generated in the `idlmembernameI` and `idlmembernameM` modules can be used by the user’s implementation code in the `idlmembernameI` module:

```c
/* 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 SEQLLEN 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 462.
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 SEQALLOC).
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:

```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` module (where `idlmembername` represents the name of the IDL member that contains the IDL definitions):

```c
/* Extract from EXAMPLET */
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` module:

```c
/* Extract from EXAMPLEM */
dcl 1 example_myseq_attr aligned like example_myseq_type;
```
3. The following example shows how the code generated in the
    idlmembernameT and idlmembernameM modules can be used by the
    user’s implementation in the idlmembernameI module:

```
/* Extract from EXAMPLE 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 445.
- “SEQLEN” on page 457.
SEQREL

Synopsis

SEQREL(in PTR sequence_control_data,
    in CHAR(*) typecode_key,
    in FIXED BIN(31) typecode_key_length)
// Frees the memory allocated to an unbounded sequence and its
// contents

Usage

Common to clients and servers.

Description

As with SEQFREE, SEQREL is used to release storage assigned to a sequence. Unlike SEQFREE, SEQREL not only clears the storage required by the sequence itself, but also that of its contents. (Storage is assigned to a sequence by calling SEQALLOC or SEQINIT.) Do not try to use the sequence again after freeing its memory, because doing so might result in a runtime error.

Parameters

The parameters for SEQREL can be described as follows:

sequence_control_data This is an in 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.

Example

Consider the following IDL:

```idl
interface example {
    typedef sequence<string> seqstr;
    attribute seqstr            myseq;
};
```

Because the contents of the preceding seqstr sequence are strings, they require dynamic allocation (for example, using STRSET).
If `SEQFREE` were used to deallocate storage associated with the preceding sequence, the dynamic contents would first need to be deallocated using `STRSET`. By contrast, `SEQREL` automatically frees the contained strings before freeing the sequence.

See also

- “SEQLOC” on page 445.
- “Memory Handling” on page 345.
SEQSET

Synopsis

SEQSET(out 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 from the element buffer area (that is, from the sequence-data parameter) into the sequence at the specified element position. 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.

**Note:** SEQSET performs a deep copy from the element buffer area to the sequence area. To avoid leaks, the element buffer area should be freed of any dynamically allocated data following a call to SEQSET.

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:

```plaintext
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 **out** parameter that contains the unbounded sequence control data.
The example can be broken down as follows:

1. Consider the following IDL:

   ```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 `idlmembernameT` module (where `idlmembername` represents the name of the IDL member that contains the IDL definitions):

   ```idl
   /* 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 `idlmembernameM` module:

   ```idl
   /* Extract from EXAMPLM */
   dcl 1 example_myseq_attr aligned like example_myseq_type;
   ```
4. The following example shows how the code generated in the
idlmodule1 and idlmodule2 modules can be used by the
user’s implementation code in the idlmodule1 module:

```idl
/* Extract from EXAMPLE 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 seqalloc(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_args.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 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 452.
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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string_pointer</td>
<td>This is an in parameter that is the unbounded string pointer containing a copy of the bounded string.</td>
</tr>
<tr>
<td>duplicate_string_pointer</td>
<td>This is an out parameter that contains the duplicated string.</td>
</tr>
</tbody>
</table>

Example

The following is an example of how to use STRDUPL in a client or server program:

```plaintext
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:

```plaintext
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 478.
- “Memory Handling” on page 345.
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:

```plaintext
TEST: PROC OPTIONS(MAIN);
   %include CORBA;
   dcl temp_string                   char(32) init('Hello there');
   dcl str_pointer             ptr;
   dcl dest                    char(64);
   call strset(str_pointer, temp_string, length(temp_string));
   call strget(str_pointer, dest, length(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 478.
**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:

```plaintext
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 |
**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.

**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`.

**Note**: `STRSET` allocates memory for the string from the program heap at runtime. See “STRFREE” on page 473 and “Unbounded Strings and Memory Management” on page 356 for details of how this memory is subsequently released.
1. Consider the following test program:

```plaintext
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:

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>THE LENGTH OF STRING 1 IS</td>
<td>16</td>
</tr>
<tr>
<td>THE LENGTH OF STRING 2 IS</td>
<td>20</td>
</tr>
</tbody>
</table>

See also

- “STRFREE” on page 473.
- “STRGET” on page 474.
- “Unbounded Strings and Memory Management” on page 356.
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 478 for more details.

Note: STSETS allocates memory for the string from the program heap at runtime. See “STRFREE” on page 473 and “Unbounded Strings and Memory Management” on page 356 for details of how this memory is subsequently released.

See also

- “STRGET” on page 474.
- “Unbounded Strings and Memory Management” on page 356.
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 z/OS 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:

```plaintext
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 */

---

corbalociiop-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 */
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:

```plaintext
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 checkErrors('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 407.
CHAPTER 9 | API Reference

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:

```plaintext
interface example {
    attribute any myany;
};
```
2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the \texttt{idlmembernameT} module (where \texttt{idlmembername} represents the name of the IDL member that contains the IDL definitions):

\begin{verbatim}
/* Extract from EXAMPLT */
dcl 1 example_myany_attr aligned,
     3 result                  ptr;
\end{verbatim}

Based on the preceding IDL, the Orbix IDL compiler also generates the following code, in the \texttt{idlmembernameM} module:

\begin{verbatim}
/* Extract from EXAMPLM */
dcl 1 example_myany_attr aligned like example_myany_type;
\end{verbatim}
3. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the idlmembername module:

```c
/* 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 389.
- “ANYSET” on page 391.
**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` module (where `idlmembername` represents the name of the IDL member that contains the IDL definitions):

   ```
   /* Extract from EXAMPLE */
   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 \texttt{idlmembernameM} module:

```c
/* Extract from EXAMPLIM */
dcl l example_myany_attr aligned like example_myany_type;
```

3. The following example shows how the code generated in the \texttt{idlmembernameI} and \texttt{idlmembernameM} modules can be used by the user's implementation code in the \texttt{idlmembernameI} module.

```c
/* 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 \texttt{CORBA::BAD_PARAM::UNKNOWN_TYPECODE} exception is raised if the typecode cannot be determined from the typecode key passed to \texttt{TYPESET}.

### See also

- “\texttt{ANYGET}” on page 389.
- “\texttt{ANYSET}” on page 391.
- “\texttt{TYPEGET}” on page 486.
**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:

```plaintext
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:

```plaintext
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
WSTRFRE(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:

```plaintext
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 500.
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:

```pli
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 500.
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:

```plaintext
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 pointer. 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 496.
# 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 500 for more details.

## See also

"WSTRGET" on page 496.
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:

```plaintext
/* 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 CHKCLIC in CICS client programs, and CHKCLIMS in IMS client programs. See Table 6 on page 56 and Table 15 on page 128 for more details of these include members.

Example

The following is an example of how to use CHECK_ERRORS in a program:

```plaintext
call strset(orig_str_ptr, input_string, length(input_string));
if check_errors('strset') ^= completion_status_yes then return;
```
Deprecated and Removed 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 MEMALLOC.
// 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 MEMBUG.
// 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 */
OBJLONO(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
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

This part contains the following appendices:

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>POA Policies</td>
<td>511</td>
</tr>
<tr>
<td>System Exceptions</td>
<td>515</td>
</tr>
<tr>
<td>Installed Data Sets</td>
<td>519</td>
</tr>
</tbody>
</table>
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.

In this appendix

This appendix contains the following sections:

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>511</td>
</tr>
<tr>
<td>POA policy listing</td>
<td>512</td>
</tr>
</tbody>
</table>

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.
Table 49 describes the policies that are supported by the Orbix PL/I runtime, and the argument used with each policy.

**Table 49: POA Policies Supported by PL/I Runtime (Sheet 1 of 3)**

<table>
<thead>
<tr>
<th>Policy</th>
<th>Argument Used</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id Assignment</td>
<td>USER_ID</td>
<td>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).</td>
</tr>
<tr>
<td>Id Uniqueness</td>
<td>MULTIPLE_ID</td>
<td>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.</td>
</tr>
<tr>
<td>Implicit Activation</td>
<td>NO_IMPLICIT_ACTIVATION</td>
<td>This policy determines the POA’s activation policy. The NO_IMPLICIT_ACTIVATION argument specifies that the POA only supports explicit activation of servants.</td>
</tr>
</tbody>
</table>
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.

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>
<thead>
<tr>
<th>Policy</th>
<th>Argument Used</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lifespan</td>
<td><code>PERSISTENT</code></td>
<td>This policy determines whether object references outlive the process in which they were created. The <code>PERSISTENT</code> 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.</td>
</tr>
</tbody>
</table>
| 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 49: POA Policies Supported by PL/I Runtime (Sheet 3 of 3)

<table>
<thead>
<tr>
<th>Policy</th>
<th>Argument Used</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Servant Retention</td>
<td>RETAIN</td>
<td>The RETAIN argument with this policy specifies that the POA retains active servants in its active object map.</td>
</tr>
<tr>
<td>Thread</td>
<td>SINGLE_THREAD_MODEL</td>
<td>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.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Exception Category</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA::INITIALIZE:: exceptions</td>
<td>page 515</td>
</tr>
<tr>
<td>CORBA::BAD_PARAM:: exceptions</td>
<td>page 516</td>
</tr>
<tr>
<td>CORBA::INTERNAL:: exceptions</td>
<td>page 516</td>
</tr>
<tr>
<td>CORBA::BAD_INV_ORDER:: exceptions</td>
<td>page 516</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNKNOWN_OPERATION</td>
<td>This exception is raised by PODEXEC, if the operation is not valid for the interface.</td>
</tr>
<tr>
<td>NO_OBJECT_IDENTIFIER</td>
<td>This exception is raised by OBJNEW, if the parameter for the object name is an invalid string.</td>
</tr>
<tr>
<td>INVALID_SERVER_NAME</td>
<td>This exception is raised if the server name that is passed does not match the server name passed to PODSRVR.</td>
</tr>
</tbody>
</table>

### CORBA::INTERNAL:: exceptions

The following exceptions are defined within the CORBA::INTERNAL:: scope:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNEXPECTED_INVOCATION</td>
<td>This exception is raised on the server side when a request is being processed, if a previous request has not completed successfully.</td>
</tr>
<tr>
<td>UNKNOWN_TYPECODE</td>
<td>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 idlmembernameX include member or the application itself.</td>
</tr>
<tr>
<td>INVALID_STREAMABLE</td>
<td>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 idlmembernameX include member of the application itself.</td>
</tr>
</tbody>
</table>

### CORBA::BAD_INV_ORDER:: exceptions

The following exceptions are defined within the CORBA::BAD_INV_ORDER:: scope:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERFACE_NOT_REGISTERED</td>
<td>This exception is raised if the specified interface has not been registered via PODREG.</td>
</tr>
<tr>
<td>INTERFACE_ALREADY_REGISTERED</td>
<td>This exception is raised by PODREG, if the client or server attempts to register the same interface more than once.</td>
</tr>
<tr>
<td>Exception Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ADAPTER_ALREADY_INITIALIZED</td>
<td>This exception is raised by ORBARGS, if it is called more than once in a client or server.</td>
</tr>
<tr>
<td>STAT_ALREADY_CALLED</td>
<td>This exception is raised by PODSTAT if it is called more than once.</td>
</tr>
<tr>
<td>SERVER_NAME_ALREADY_SET</td>
<td>This exception is raised by PODSRVR, if the API is called more than once.</td>
</tr>
<tr>
<td>SERVER_NAME_NOT_SET</td>
<td>This exception is raised by OBJNEW, PODREQ, OBJGTD, or PODRUN, if PODSRVR is called.</td>
</tr>
<tr>
<td>NO_CURRENT_REQUEST</td>
<td>This exception is raised by PODREQ, if no request is currently in progress.</td>
</tr>
<tr>
<td>ARGS_NOT_READ</td>
<td>This exception is raised by PODPUT, if the in or inout parameters for the request have not been processed.</td>
</tr>
<tr>
<td>ARGS_ALREADY_READ</td>
<td>This exception is raised by PODGET, if the in or inout parameters for the request have already been processed.</td>
</tr>
<tr>
<td>TYPESET_NOT_CALLED</td>
<td>This exception is raised by ANYSET or TYPEGET, if the typecode for the any type has not been set via a call to TYPESET.</td>
</tr>
</tbody>
</table>
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 | page 519 |
| List of PL/I-related data sets | page 519 |

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 50 lists the installed data sets that are relevant to PL/I.

Table 50: List of Installed Data Sets Relevant to PL/I (Sheet 1 of 4)

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orbixhlq.ADMIN.GRAMMAR</td>
<td>Contains itadmin grammar files.</td>
</tr>
<tr>
<td>orbixhlq.ADMIN.HELP</td>
<td>Contains itadmin help files.</td>
</tr>
<tr>
<td>orbixhlq.ADMIN.LOADLIB</td>
<td>Contains Orbix administration programs.</td>
</tr>
<tr>
<td>orbixhlq.CONFIG</td>
<td>Contains Orbix configuration information.</td>
</tr>
<tr>
<td>Data Set</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>orbixhlq.DEOMS.ARTIX.BLD.JCLLIB</td>
<td>Contains jobs to build the Artix Transport demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEOMS.CICS.MFAMAP</td>
<td>Used to store CICS server adapter mapping member information for demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEOMS.CICS.PLI.BLD.JCLLIB</td>
<td>Contains jobs to build the CICS PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEOMS.CICS.PLI.LOADLIB</td>
<td>Used to store programs for the CICS PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEOMS.CICS.PLI.PLINCL</td>
<td>Used to store generated files for the CICS PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEOMS.CICS.PLI.README</td>
<td>Contains documentation for the CICS PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEOMS.CICS.PLI.SRC</td>
<td>Contains program source for the CICS PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEOMS.IDL</td>
<td>Contains IDL for demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEOMS.IMS.MFAMAP</td>
<td>Used to store IMS server adapter mapping member information for demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEOMS.IMS.PLI.BLD.JCLLIB</td>
<td>Contains jobs to build the IMS PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEOMS.IMS.PLI.LOADLIB</td>
<td>Used to store programs for the IMS PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEOMS.IMS.PLI.PLINCL</td>
<td>Used to store generated files for the IMS PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEOMS.IMS.PLI.README</td>
<td>Contains documentation for the IMS PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEOMS.IMS.PLI.SRC</td>
<td>Contains program source for the IMS PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEOMS.IORs</td>
<td>Used to store IORs for demonstrations.</td>
</tr>
<tr>
<td>Data Set</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.PLI.BLD.JCLLIB</td>
<td>Contains jobs to build the PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.PLI.LOADLIB</td>
<td>Used to store programs for the PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.PLI.MAP</td>
<td>Used to store name substitution maps for the PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.PLI.PLINCL</td>
<td>Used to store generated files for the PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.PLI.README</td>
<td>Contains documentation for the PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.PLI.RUN.JCLLIB</td>
<td>Contains jobs to run the PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.PLI.SRC</td>
<td>Contains program source for the PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.DEMOS.TYPEINFO</td>
<td>Optional type information store.</td>
</tr>
<tr>
<td>orbixhlq.DOMAINS</td>
<td>Contains Orbix configuration information.</td>
</tr>
<tr>
<td><a href="mailto:orbixhlqINCLUDE.IT@CICS.IDL">orbixhlqINCLUDE.IT@CICS.IDL</a></td>
<td>Contains IDL files.</td>
</tr>
<tr>
<td><a href="mailto:orbixhlqINCLUDE.IT@IMS.IDL">orbixhlqINCLUDE.IT@IMS.IDL</a></td>
<td>Contains IDL files.</td>
</tr>
<tr>
<td><a href="mailto:orbixhlqINCLUDE.IT@MFA.IDL">orbixhlqINCLUDE.IT@MFA.IDL</a></td>
<td>Contains IDL files.</td>
</tr>
<tr>
<td>orbixhlqINCLUDE.OMG.IDL</td>
<td>Contains IDL files.</td>
</tr>
<tr>
<td>orbixhlqINCLUDE.ORBIX.IDL</td>
<td>Contains IDL files.</td>
</tr>
<tr>
<td><a href="mailto:orbixhlqINCLUDE.ORBIX@XT.IDL">orbixhlqINCLUDE.ORBIX@XT.IDL</a></td>
<td>Contains IDL files.</td>
</tr>
<tr>
<td>orbixhlqINCLUDE.PLINCL</td>
<td>Contains include files for PL/I demonstrations.</td>
</tr>
<tr>
<td>orbixhlq.JCLLIB</td>
<td>Contains jobs to run Orbix.</td>
</tr>
<tr>
<td>orbixhlq.LKED</td>
<td>Contains side-decks for the DLLs.</td>
</tr>
<tr>
<td>orbixhlq.LOADLIB</td>
<td>Contains binaries &amp; DLLs.</td>
</tr>
<tr>
<td>orbixhlq.LPALIB</td>
<td>Contains LPA eligible programs.</td>
</tr>
</tbody>
</table>
### Table 50: List of Installed Data Sets Relevant to PL/I (Sheet 4 of 4)

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orbixhlq.MFA.LOADLIB</td>
<td>Contains DLLS required for deployment of Orbix programs in IMS.</td>
</tr>
<tr>
<td>orbixhlq.PLL.OBJLIB</td>
<td>Contains programs for Orbix PL/I support.</td>
</tr>
<tr>
<td>orbixhlq.PLICICS.OBJLIB</td>
<td>Contains programs for CICS-to-CICS PL/I support.</td>
</tr>
<tr>
<td>orbixhlq.PROCLIB</td>
<td>Contains JCL procedures.</td>
</tr>
</tbody>
</table>
ORXCOPY Utility

This appendix provides details of the ORXCOPY utility which allows you to copy data between different types of files, such as on-host data sets and UNIX-based HFS files.

In this appendix

This appendix contains the following sections:

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsis</td>
<td>524</td>
</tr>
<tr>
<td>Description</td>
<td>524</td>
</tr>
<tr>
<td>Operands</td>
<td>524</td>
</tr>
<tr>
<td>UNIX examples</td>
<td>525</td>
</tr>
<tr>
<td>JCL example</td>
<td>525</td>
</tr>
<tr>
<td>Restriction</td>
<td>525</td>
</tr>
</tbody>
</table>
Synopsis

```
norcop in-file out-file
```

Description

The ORXCOPY utility is used to transfer data between different types of MVS files, in particular between record-oriented data sets and stream-oriented UNIX files.

Multiple records are treated as a single line if they contain a backslash (that is, `\`) in the continuation column. The continuation column is the last column in a variable-length record (VB) or the ninth-to-last column in a fixed-length record (FB). The final eight columns in an FB data set are reserved for sequence numbers and are ignored.

The ORXCOPY utility allows data to be transferred back and forth with little or no loss of information. When HFS files with long lines are copied into FB or VB data sets with shorter record lengths, the lines are wrapped across multiple records using the continuation column. When files are copied between data sets of different record lengths, lines are "unwrapped" and "re-wrapped" as necessary.

Most kinds of files used in Orbix (for example, license files, IDL files, configuration files, and C++ files) are equivalent in both "wrapped" and "unwrapped" form.

Operands

The `in-file` and `out-file` qualifiers for ORXCOPY represent the names of MVS files or data sets. The following rules apply:

- Names beginning with "DD:" or "//DD:" are assumed to refer to an allocated DD statement.
- Other names beginning with "//" and not containing additional "/" characters are assumed to be data sets.
- Single quotes indicate a dataset name (however, ORXCOPY does not attempt to infer a "prefix" qualifier such as the user name).
- Names that might refer to either an MVS data set or an HFS file should be specified unambiguously with an appropriate prefix. For example:

```
//README.TXT  Data set
./README.TXT  HFS file
```
UNIX examples

The following command copies a domain configuration from a PDS member to a UNIX file:

```
orxcopy "//HLQ.ORBIX62.DOMAINS(FILDEDMA)" filedomain.cfg
```

The following command copies a C++ source file into a PDS:

```
orxcopy objectImpl.h "//HLQ.PROJECT.H(IMPL)"
```

The following command reads an IOR stored in a PDS:

```
orxcopy "//HLQ.ORBIX62.DEMOS.IORS(EXTENDED)" extend.ior
```

JCL example

The following piece of JCL copies a license file from a VB data set to an FB PDS:

```
//GO  EXEC PROC=ORXG,PGM=ORXCOPY,
//   PPARM="DD:IN DD:OUT(LICENSES)"
//IN  DD DISP=SHR,DSN=MY.FTPED.LICENSE.FILE
//OUT DD DISP=SHR,DSN=HLQ.ORBIX62.CONFIG
```

Restriction

The ORXCOPY utility does not support a file specification of "DD:NAME" where NAME represents a DD card that uses the "PATH=" keyword. You must specify the pathname directly to ORXCOPY instead.
Index

A
abstract interfaces in IDL 212
ADAPTER_ALREADY_INITIALIZED exception 517
address space layout for PL/I batch application 47
ANYFREE function 387
ANYGET function 389
ANYSET function 391
any type
   in IDL 216
   mapping to PL/I 262
   memory handling for 364
APIs 377
application interfaces, developing 22, 59, 131
ARGS_ALREADY_READ exception 517
ARGS_ALREADY_READ_EXCEPTION 517
array type
   in IDL 225
   mapping to PL/I 261
attributes
   in IDL 201
   mapping to PL/I 275

B
basic types
   in IDL 214
   mapping to PL/I 241
   bitwise operators 232
   boolean type, mapping to PL/I 245
bounded sequences
   mapping to PL/I 258
   memory handling for 347
built-in types in IDL 214

C
char type
   in IDL 215
   mapping to PL/I 247
CHECK ERRORS function 503
CHKCICS include member 128
CHKCLCIC include member 128
CHKCLAIMS include member 128
CHKKERRS include member 56
CHKKERRS copybook 18
client output for batch 46
clients
   building for batch 41
   building for CICS 161
   building for CICS two-phase commit 181
   building for IMS 88, 109, 110, 181, 182
   building for IMS two-phase commit 109
   introduction to 7
   preparing to run in CICS 162
   preparing to run in CICS for two-phase commit 183
   preparing to run in IMS 89
   preparing to run in IMS for two-phase commit 111
   running in batch 45
   writing for batch 38
   writing for CICS 156
   writing for CICS two-phase commit 167
   writing for IMS 84
   writing for IMS two-phase commit 94
configuration domains 11
constant definitions in IDL 229
constant expressions in IDL 232
constant fixed types in IDL 219
CORBA
   introduction to 4
   objects 5
   CORBACOM copybook 19
   CORBACOM include member 56, 128
   CORBA copybook 19
   CORBA include member 56, 128
   CORBASV copybook 19
   CORBASV include member 56, 128

D
data sets installed 519, 523
data types, defining in IDL 228
decimal fractions 219
DISPINIT copybook 19
DISPINIT include member 56, 128
DLIDATA include member 56
E
empty interfaces in IDL 203
enum type
  in IDL 221
  mapping to PL/I 246
  ordinal values of 221
exceptions, in IDL 202
  See also system exceptions, user exceptions
EXCNAME copybook 19
EXCNAME include member 57, 129
extended built-in types in IDL 217

F
fixed type
  in IDL 218
  mapping to PL/I 251
floating point type in IDL 215
forward declaration of interfaces in IDL 209

G
GETUNIQ include member 57

I
Id Assignment policy 512
identifier names, mapping to PL/I 237, 239
IDL
  abstract interfaces 212
  arrays 225
  attributes 201
  built-in types 214
  constant definitions 229
  constant expressions 232
defining 23, 59, 131
empty interfaces 203
enum type 221
exceptions 202
extended built-in types 217
forward declaration of interfaces 209
inheritance redefinition 208
interface inheritance 204
introduction to interfaces 5
local interfaces 210
modules and name scoping 195
multiple inheritance 205
object interface inheritance 207
operations 199
sequence type 226
struct type 222
structure 194
union type 223
valuetypes 211
IDL-to-PL/I mapping
  any type 262
  array type 261
  attributes 275
  basic types 241
  boolean type 245
  char type 247
  enum type 246
  exception type 264
  fixed type 251
  identifier names 237, 239
  octet type 247
  operations 270
  sequence type 258
  string type 248
  struct type 254
typedefs 268
union type 255
user exception type 264
Id Uniqueness policy 512
IIOP protocol 4
Implicit Activation policy 512
IMSPCB include member 57
include members, generating for batch 24
include members, generating for CICS 137
include members, generating for IMS 65
inheritance redefinition in IDL 208
INTERFACE_ALREADY_REGISTERED
  exception 516
  interface inheritance in IDL 204
INTERFACE_NOT_REGISTERED exception 516
interfaces, developing for your application 22, 59, 131
INVALID_SERVER_NAME exception 516
INVALID_STREAMABLE exception 516
IORREC copybook 19

J
JCL components, checking 21, 58, 130

L
Lifespan policy 513
local interfaces in IDL 210
local object pseudo-operations 211
location domains 11
locator daemon
  introduction to 12
  starting 43
long double type in IDL 218
long long type in IDL 217

M
MEMALLOC function 393
MEMDEBUG function 394
MEMFREE function 396
memory handling
  any type 364
  bounded sequences 347
  object references 360
  routines for 371
  unbounded sequences 351
  unbounded strings 356
  user exceptions 369
modules and name scoping in IDL 195
MULTIPLE_ID argument 512
multiple inheritance in IDL 205

N
NO_CURRENT_REQUEST exception 517
node daemon
  introduction to 12
  starting 44
NO_IMPLICIT_ACTIVATION argument 512
NO_OBJECT_IDENTITY exception 516

O
OBJ2STR function 407
  in batch server mainline 35
OBJDUPL function 397
object interface inheritance in IDL 207
object references
  introduction to 5
  memory handling for 360
object request broker. See ORB
objects, defined in CORBA 5
OBJGTID function 399
OBJNEW function 401
  in batch server mainline 35
  in CICS server mainline 149
OBJNEW function in IMS server mainline 77
OBJREL function
  in batch client 40
  in batch server mainline 35
  in CICS client 160
  in CICS server mainline 149
  in IMS client 87
OBJIREL function in IMS server mainline 77
OBJIRIR function 405
octet type
  in IDL 215
  mapping to PL/I 247
operations
  in IDL 199
  mapping to PL/I 270
ORB, role of 7
ORBARGS function 409
  in batch client 40
  in batch server mainline 35
  in CICS client 159
  in CICS server mainline 148
  in IMS client 87, 107, 178
ORBARGS function in IMS server mainline 77
Orbix IDL compiler
  configuration settings 331
    -D argument 311
    -E argument 312
    introduction to 24, 62, 134
    -L argument 314
    -M argument 316
    -O argument 323
    running 294
    -S argument 325
    specifying arguments for 308
    -T argument 326
    -V argument 329
    -W argument 330
Orbix locator daemon. See locator daemon
Orbix node daemon. See node daemon
Orbix PL/I runtime 48, 377
ORBREG function
  in IMS client 107, 178
ORBSTAT function
  in IMS client 106, 178
ORXCOPY utility 523

P
PERSISTENT argument 513
PL/I runtime 48
PL/I source, generating for batch 24
PL/I source, generating for CICS 137
PL/I source, generating for IMS 65
PL/I structures 27, 65, 137
plug-ins, introduction to 9
PODERR function 413
PODEXEC function 418
  in batch client 40
  in CICS client 160
  in IMS client 87
PODGET function 259, 421
  in batch server implementation 31
PODGET function in CICS server implementation 146
PODGET function in IMS server implementation 74
PODINFO function 424
PODPUT function 260, 426
  in batch server implementation 32
PODPUT function in CICS server implementation 146
PODPUT function in IMS server implementation 74
PODREG function 429
  in batch client 40
  in batch server mainline 35
  in CICS client 159
  in CICS server mainline 149
  in IMS client 87
PODREG function in IMS server mainline 77
PODREQ function 431
  in batch server implementation 31
  in CICS server implementation 145
PODREQ function in IMS server implementation 74
PODRUN function 434
  in batch server mainline 35
  in CICS server mainline 149
PODRUN function in IMS server mainline 77
PODSRVR function 435
  in batch server mainline 35
  in CICS server mainline 148
PODSRVR function in IMS server mainline 77
PODSTAT function 437
  in batch server mainline 35
  in CICS client 159
  in CICS server mainline 148
  in IMS client 86
PODSTAT function in IMS server mainline 76
PODTIME function 440
PODTXNB function 442
PODTXNE function 443

R
READIOR copybook 19

Request Processing policy 513
RETAIN argument 514

S
SEQALLOC function 260
SEQALOC function 445
SEQDUPL function 448
SEQFREE function 450
SEQGET function 452
SEQINIT function 455
SEQLEN function 457
SEQSET function 459
SEQMAX function 462
SEQSET function 467
sequence type
  in IDL 226
  mapping to PL/I 258
  See also memory handling
Servant Retention policy 514
SERVER_NAME_ALREADY_SET exception 517
SERVER_NAME_NOT_SET exception 517
server output for batch 46
servers
  building for batch 36
  building for CICS 150
  building for CICS two-phase commit 182
  building for IMS 78
  building for IMS two-phase commit 110
  introduction to 7
  preparing to run in CICS 151
  preparing to run in IMS 79
  running in batch 45
  writing batch implementation code for 30
  writing batch mainline code for 33
  writing CICS implementation code for 144
  writing CICS mainline code for 147
  writing IMS implementation code for 72
  writing IMS mainline code for 75
SETUPCL copybook 20
SETUPSV copybook 20
SIMPLIDL 296
SIMPLIDL JCL
  example for CICS 134
  example for IMS 62
SINGLE_THREAD_MODEL argument 514
SSL 9
STAT_ALREADY_CALLED exception 517
STR2OBJ function 481
  in CICS client 160
INDEX

in IMS client 87
STR2OBJ function in batch client 40
STRING function 470
STRDUPL function 472
STRGET function 474
in batch server implementation 31
STRGET function in CICS server implementation 145
STRGET function in IMS server implementation 74
string type
  in IDL 215
  mapping to PL/I 248
See also memory handling
STRING function 476
STRSET function 478
  in CICS client 159
  in IMS client 87, 107, 178
STRSETS function 480
STRTOOBJ function
  in IMS client 87, 178
struct type
  in IDL 222
  mapping to PL/I 254

T
Thread policy 514
two-phase commit
  building C++ servers for 110, 182
  building CICS clients for 181
  building IMS clients for 109
  preparing clients to run in CICS for 183
  preparing clients to run in IMS for 111
  running CICS client against batch servers 189
  running IMS client against batch servers 117
  writing CICS clients for 167
  writing IMS clients for 94
typedef, mapping to PL/I 268
TYPEGET function 486
TYPESET function 489
TYPESET_NOT_CALLED exception 517

U
unbounded sequences, memory handling for 351
unbounded strings, memory handling for 356
UNEXPECTED_INVOCATION exception 516
union type
  in IDL 223
  mapping to PL/I 255

UNKNOWN exception 515
UNKNOWN_OPERATION exception 516
UNKNOWN_TYPECODE exception 516
URLSTR copybook 20
URLSTR include member 57, 129
USE_ACTIVE_OBJECT_MAP_ONLY argument 513
user exceptions
  mapping to PL/I 264
    memory handling for 369
  USER_ID argument 512

V
valuetypes in IDL 211

W
wchar type in IDL 218
WSTRCON function 491
WSTRDUP function 493
WSTRGET function 496
wstring type in IDL 218
WSTRLEN function 498
WSTRSET function 500
WSTRSTS function 502