



# Orbix E2A APPLICATION SERVER PLATFORM™

## COBOL Programmer's Guide and Reference

Version 5.0.1, July 2002

End to end is nothing.  
END 2 ANYWHERE *is everything.*™

Orbix, Orbix E2A, Orbix E2A Application Server Platform, Orbix E2A Application Server, Orbix E2A XMLBus, Orbix E2A IMS Connector, Adaptive Runtime Technology, are trademarks or registered trademarks of IONA Technologies PLC and/or its subsidiaries.

IONA, IONA Technologies, the IONA logo, End 2 Anywhere, End To Anywhere, IONA e-Business Platform, and Total Business Integration are trademarks or registered trademarks of IONA Technologies PLC and/or its subsidiaries.

Java and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the United States and other countries. All other trademarks that appear herein are the property of their respective owners.

While the information in this publication is believed to be accurate, IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA Technologies PLC shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

---

#### COPYRIGHT NOTICE

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of IONA Technologies PLC. No third party intellectual property right liability is assumed with respect to the use of the information contained herein. IONA Technologies PLC assumes no responsibility for errors or omissions contained in this book. This publication and features described herein are subject to change without notice.

Copyright © 1998, 2002 IONA Technologies PLC. All rights reserved.

All products or services mentioned in this manual are covered by the trademarks, service marks, or product names as designated by the companies who market those products.

M 2 9 0 9

# Contents

<b>List of Figures</b>	<b>ix</b>
<b>List of Tables</b>	<b>xi</b>
<b>Preface</b>	<b>xiii</b>
<b>Chapter 1 Introduction to Orbix E2A</b>	<b>1</b>
<b>Why CORBA?</b>	<b>2</b>
CORBA Objects	3
Object Request Broker	5
<b>CORBA Application Basics</b>	<b>6</b>
<b>Orbix Plug-In Design</b>	<b>7</b>
<b>Orbix Application Deployment</b>	<b>9</b>
Location Domains	10
Configuration Domains	11
<b>Chapter 2 Getting Started in Batch</b>	<b>13</b>
<b>Overview and Setup Requirements</b>	<b>14</b>
<b>Developing the Application Interfaces</b>	<b>19</b>
Defining IDL Interfaces	20
Generating COBOL Source and Copybooks	21
<b>Developing the Server</b>	<b>24</b>
Writing the Server Implementation	25
Writing the Server Mainline	28
Building the Server	34
<b>Developing the Client</b>	<b>35</b>
Writing the Client	36
Building the Client	41
<b>Running the Application</b>	<b>42</b>
Starting the Orbix E2A Locator Daemon	43
Starting the Orbix E2A Node Daemon	44
Running the Server and Client	45
Application Output	46

<b>Application Address Space Layout</b>	<b>47</b>
<b>Chapter 3 Getting Started in IMS</b>	<b>49</b>
<b>Overview and Setup Requirements</b>	<b>50</b>
<b>Developing the Application Interfaces</b>	<b>56</b>
Defining IDL Interfaces	57
Generating COBOL Source and Copybooks	58
<b>Developing the Server</b>	<b>62</b>
Writing the Server Implementation	63
Writing the Server Mainline	67
Building the Server	72
Preparing the Server to Run in IMS	73
<b>Developing and Running the Client</b>	<b>76</b>
Writing the Client	77
Building and Running the Client	82
<b>Chapter 4 Getting Started in CICS</b>	<b>83</b>
<b>Overview and Setup Requirements</b>	<b>84</b>
<b>Developing the Application Interfaces</b>	<b>89</b>
Defining IDL Interfaces	90
Generating COBOL Source and Copybooks	91
<b>Developing the Server</b>	<b>95</b>
Writing the Server Implementation	96
Writing the Server Mainline	100
Building the Server	104
Preparing the Server to Run in CICS	105
<b>Developing and Running the Client</b>	<b>109</b>
Writing the Client	110
Building and Running the Client	115
<b>Chapter 5 IDL Interfaces</b>	<b>117</b>
<b>IDL</b>	<b>118</b>
<b>Modules and Name Scoping</b>	<b>119</b>
<b>Interfaces</b>	<b>120</b>
Interface Contents	122
Operations	123
Attributes	126
Exceptions	127

Empty Interfaces	128
Inheritance of Interfaces	129
Multiple Inheritance	130
Inheritance of the Object Interface	132
Inheritance Redefinition	133
Forward Declaration of IDL Interfaces	134
Local Interfaces	135
Valuetypes	137
Abstract Interfaces	139
<b>IDL Data Types</b>	<b>140</b>
Built-in Data Types	141
Extended Built-in Data Types	143
Complex Data Types	146
Enum Data Type	147
Struct Data Type	148
Union Data Type	149
Arrays	151
Sequence	152
Pseudo Object Types	153
<b>Defining Data Types</b>	<b>154</b>
Constants	155
Constant Expressions	158
 <b>Chapter 6 IDL-to-COBOL Mapping</b>	 <b>161</b>
Mapping for Identifier Names	163
Mapping for Type Names	165
Mapping for Basic Types	166
Mapping for Boolean Type	171
Mapping for Enum Type	174
Mapping for Char Type	177
Mapping for Octet Type	179
Mapping for String Types	181
Mapping for Wide String Types	186
Mapping for Fixed Type	187
Mapping for Struct Type	191
Mapping for Union Type	193
Mapping for Sequence Types	198
Mapping for Array Type	203
Mapping for the Any Type	205

Mapping for User Exception Type	207
Mapping for Typedefs	210
Mapping for the Object Type	213
Mapping for Constant Types	215
Mapping for Operations	218
Mapping for Attributes	223
Mapping for Operations with a Void Return Type and No Parameters	224
Mapping for Inherited Interfaces	226
Mapping for Multiple Interfaces	233
<b>Chapter 7 Orbix E2A IDL Compiler</b>	<b>237</b>
Running the Orbix E2A IDL Compiler	238
Generated COBOL Members	240
Orbix E2A IDL Compiler Arguments	242
Specifying the Compiler Arguments	243
-M Argument	244
-O Argument	250
-Q Argument	252
-S Argument	253
-T Argument	254
-Z Argument	256
Configuration Member Settings	257
COBOL Configuration Settings	258
Adapter Mapping Member Configuration Settings	261
<b>Chapter 8 API Reference</b>	<b>263</b>
API Reference Summary	264
API Reference Details	268
ANYFREE	270
ANYGET	272
ANYSET	274
COAERR	277
COAGET	281
COAPUT	286
COAREQ	292
COARUN	297
MEMALLOC	298
MEMFREE	300

OBJDUP	301
OBJGETID	303
OBJNEW	305
OBJREL	308
OBJRIR	310
OBJTOSTR	312
ORBARGS	314
ORBEXEC	317
ORBHOST	323
ORBREG	325
ORBSRVR	328
ORBSTAT	329
ORBTIME	333
SEQALLOC	335
SEQDUP	339
SEQFREE	344
SEQGET	347
SEQSET	350
STRFREE	355
STRGET	357
STRLEN	360
STRSET	362
STRSETP	365
STRTOOBJ	367
TYPEGET	369
TYPESET	371
WSTRFREE	373
WSTRGET	374
WSTRLEN	375
WSTRSET	376
WSTRSETP	377
CHECK-STATUS	378
<b>Deprecated APIs</b>	<b>381</b>

<b>Chapter 9</b>	<b>Memory Handling</b>	<b>383</b>
	<b>Operation Parameters</b>	<b>384</b>
	Unbounded Sequences and Memory Management	385
	Unbounded Strings and Memory Management	389
	Object References and Memory Management	393
	The any Type and Memory Management	397
	User Exceptions and Memory Management	402
	<b>Memory Management Routines</b>	<b>404</b>
<b>Chapter 10</b>	<b>POA Policies</b>	<b>407</b>
<b>Chapter 11</b>	<b>System Exceptions</b>	<b>411</b>
	<b>Index</b>	<b>415</b>



# List of Figures

Figure 1: The Nature of Abstract CORBA Objects	3
Figure 2: The Object Request Broker	5
Figure 3: Address Space Layout for an Orbix E2A COBOL Application	47
Figure 4: Inheritance Hierarchy for PremiumAccount Interface	131

LIST OF FIGURES

# List of Tables

Table 1: Supplied Code and JCL	15
Table 2: Supplied Copybooks	16
Table 3: Generated Server Source Code Members	21
Table 4: Generated COBOL Copybooks	22
Table 5: Supplied Code and JCL	51
Table 6: Supplied Copybooks	53
Table 7: Generated Server Source Code Members	59
Table 8: Generated COBOL Copybooks	60
Table 9: Generated IMS Adapter Mapping Member	61
Table 10: Supplied Code and JCL	85
Table 11: Supplied Copybooks	87
Table 12: Generated Server Source Code Members	92
Table 13: Generated COBOL Copybooks	93
Table 14: Generated CICS Adapter Mapping Member	94
Table 15: CORBA::LocalObject Pseudo-Operations and Return Values	136
Table 16: Built-in IDL Data Types, Sizes, and Values	141
Table 17: Extended built-in IDL Data Types, Sizes, and Values	143
Table 18: Mapping for Basic IDL Types	166
Table 19: COBOL Members Generated by the Orbix E2A IDL Compiler	240
Table 20: Example of Default Generated Data Names	245
Table 21: Example of Level-0-Scoped Generated Data Names	247
Table 22: Example of Level-1-Scoped Generated Data Names	247
Table 23: Example of Level-2-Scoped Generated Data Names	248
Table 24: COBOL Configuration Settings	259
Table 25: Adapter Mapping Member Configuration Settings	262
Table 26: Summary of Common Services and Their COBOL Identifiers	310

## LIST OF TABLES

Table 27: Memory Handling for IN Unbounded Sequences	385
Table 28: Memory Handling for INOUT Unbounded Sequences	386
Table 29: Memory Handling for OUT and Return Unbounded Sequences	387
Table 30: Memory Handling for IN Unbounded Strings	389
Table 31: Memory Handling for INOUT Unbounded Strings	390
Table 32: Memory Handling for OUT and Return Unbounded Strings	391
Table 33: Memory Handling for IN Object References	393
Table 34: Memory Handling for INOUT Object References	394
Table 35: Memory Handling for OUT and Return Object References	395
Table 36: Memory Handling for IN Any Types	397
Table 37: Memory Handling for INOUT Any Types	398
Table 38: Memory Handling for OUT and Return Any Types	400
Table 39: Memory Handling for User Exceptions	402
Table 40: POA Policies Supported by the Orbix E2A COBOL Runtime	408

# Preface

Orbix E2A is a full implementation from IONA Technologies of the Common Object Request Broker Architecture (CORBA), as specified by the Object Management Group (OMG). Orbix E2A complies with the following specifications:

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

The Orbix E2A Application Server Platform Mainframe Edition is IONA's implementation of the CORBA standard for the OS/390 platform. Orbix E2A Application Server Platform Mainframe Edition documentation is periodically updated. New versions between release are available at <http://www.iona.com/docs>.

If you need help with this or any other IONA products, contact IONA at [support@iona.com](mailto:support@iona.com). Comments on IONA documentation can be sent to [doc-feedback@iona.com](mailto:doc-feedback@iona.com).

---

## Audience

This guide is intended for COBOL application programmers who want to develop Orbix E2A applications in a native OS/390 environment.

---

## Supported Compilers

The supported compilers are:

- IBM COBOL for OS/390 & VM version 2.1.1.
- IBM COBOL for OS/390 & VM version 2.1.2.
- IBM COBOL for OS/390 & VM version 2.2.1.

## Related Documentation

The related documentation provided includes:

- The *PL/I Programmer's Guide and Reference*, which provides details about developing, in a native OS/390 environment, Orbix E2A PL/I 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 E2A applications in C++ in various environments, including OS/390.
- The *Mainframe Migration Guide*, which provides details of migration issues for users who have migrated from IONA's Orbix 2.3-based solution for OS/390 to Orbix E2A Application Server Platform Mainframe Edition.

## Organization of this Guide

This guide is divided as follows:

### [Chapter 1, "Introduction to Orbix E2A"](#)

This chapter provides an introductory overview of CORBA and Orbix E2A.

### [Chapter 2, "Getting Started in Batch"](#)

This chapter introduces batch application programming with Orbix E2A, by showing how to use Orbix E2A to develop a simple distributed application that features a COBOL client and server, each running in batch in its own region.

### [Chapter 3, "Getting Started in IMS"](#)

This chapter introduces IMS application programming with Orbix E2A, by showing how to use Orbix E2A to develop a simple distributed application that features a COBOL client running in batch and a COBOL server running in IMS.

### [Chapter 4, "Getting Started in CICS"](#)

This chapter introduces CICS application programming with Orbix E2A, by showing how to use Orbix E2A to develop a simple distributed application that features a COBOL client running in batch and a COBOL server running in CICS.

**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-COBOL 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-COBOL mapping rules and shows, by example, how each IDL type is represented in COBOL.

**Chapter 7, “Orbix E2A IDL Compiler”**

This chapter describes the Orbix E2A IDL compiler in terms of the JCL used to run it, the COBOL members that it creates, the arguments that you can use with it, and the configuration settings that it uses.

**Chapter 8, “API Reference”**

This chapter summarizes the API functions that are defined for the Orbix E2A COBOL runtime, in pseudo-code. It explains how to use each function, with an example of how to call it from COBOL.

**Chapter 9, “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 E2A COBOL application. It first describes in detail the memory handling rules adopted by the COBOL 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.

**Chapter 10, “POA Policies”**

This chapter summarizes the POA policies that are supported by the Orbix E2A COBOL runtime, and the argument used with each policy.

**Chapter 11, “System Exceptions”**

This chapter summarizes the Orbix E2A system exceptions that are specific to the Orbix E2A COBOL runtime

## Additional Related Resources

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

<http://www.iona.com/support/kb/>

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

<http://www.iona.com/support/update/>

## Typographical Conventions

This guide uses the following typographical conventions:

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

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

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

**Italic**      Italic words in normal text represent *emphasis* and *new terms*.

Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example:

```
% cd /users/your_name
```

**Note:** Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.



## Keying Conventions

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS, Windows NT, Windows 95, or Windows 98 command prompt.
. . . . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

## PREFACE

# Introduction to Orbix E2A

*With Orbix E2A, you can develop and deploy large-scale enterprise-wide CORBA systems in languages such as COBOL, PL/I, C++, and Java. Orbix E2A 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 E2A Application Server Platform Mainframe Edition is IONA's CORBA solution for the OS/390 environment.*

---

## In this chapter

This chapter discusses the following topics:

<a href="#">Why CORBA?</a>	<a href="#">page 2</a>
<a href="#">CORBA Application Basics</a>	<a href="#">page 6</a>
<a href="#">Orbix Plug-In Design</a>	<a href="#">page 7</a>
<a href="#">Orbix Application Deployment</a>	<a href="#">page 9</a>

---

# 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 COBOL, PL/I, 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

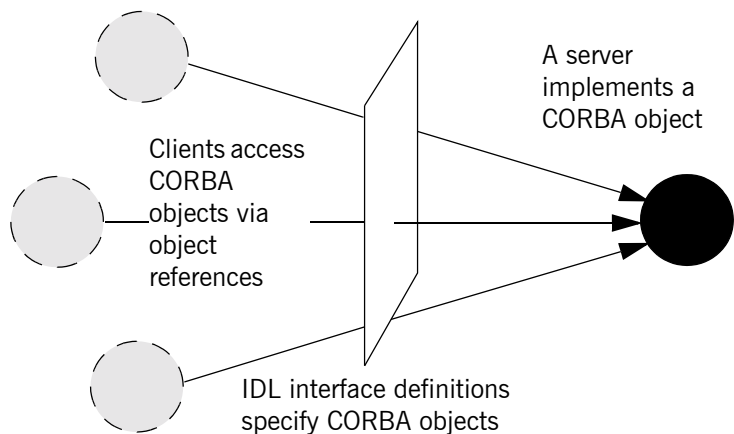
CORBA solutions are available for every common environment and are used to integrate applications written in C, C++, Java, Ada, Smalltalk, COBOL, and PL/I running on embedded systems, PCs, UNIX hosts, and mainframes. CORBA objects running in these environments can cooperate seamlessly. Through COMet, 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 COBOL, PL/I, 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

---

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, it means that you can change the programming language in which an object is implemented without affecting the clients that access the object. It also means that you can make existing objects available across a distributed network.

---

# Object Request Broker

## Overview

---

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

---

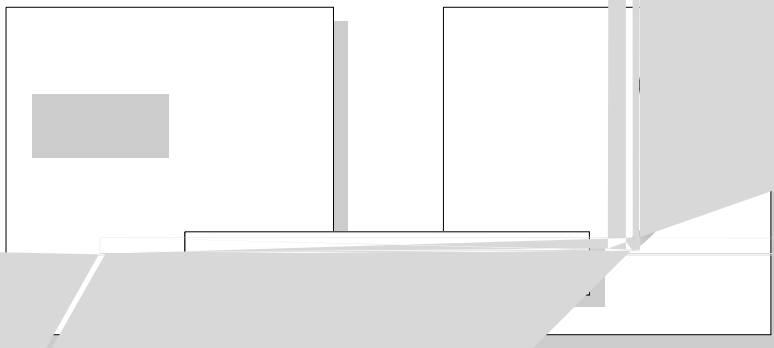
## Role of an ORB

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 on page 6](#), 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.

**Graphical overview of ORB role**

**Figure 2** provides a graphical overview of the role of the ORB in network communications.





---

# 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 COBOL, PL/I, C++, or Java from IDL definitions. Generated COBOL and PL/I consists of *server skeleton code*, which you use to implement CORBA objects.

---

## Client invocations on CORBA objects

When an Orbix E2A COBOL client on OS/390 calls a member function on a CORBA object on another platform, the call is transferred through the COBOL 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 E2A COBOL server object on OS390, the ORB passes the function call through the COBOL 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 COBOL, PL/I, 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:

<a href="#">Location Domains</a>	<a href="#">page 10</a>
<a href="#">Configuration Domains</a>	<a href="#">page 11</a>

---

# Location Domains

---

## Overview

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

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

---

## Locator daemon

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

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

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

---

## Node daemon

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

---

## Configuration Domains

### Overview

---

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

---

### Plug-in design

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

# Getting Started in Batch

*This chapter introduces batch application programming with Orbix E2A, by showing how to use Orbix E2A to develop a simple distributed application that features a COBOL client and server, each running in its own region.*

## In this chapter

This chapter discusses the following topics:

<a href="#">Overview and Setup Requirements</a>	<a href="#">page 14</a>
<a href="#">Developing the Application Interfaces</a>	<a href="#">page 19</a>
<a href="#">Developing the Server</a>	<a href="#">page 24</a>
<a href="#">Developing the Client</a>	<a href="#">page 35</a>
<a href="#">Running the Application</a>	<a href="#">page 42</a>
<a href="#">Application Address Space Layout</a>	<a href="#">page 47</a>

**Note:** The example provided in this chapter does not reflect a real-world scenario that requires the Orbix E2A Application Server Platform Mainframe Edition, because the supplied client and server are written in COBOL and running on OS/390. The example is supplied to help you quickly familiarize with the concepts of developing a batch COBOL application with Orbix E2A.

---

# Overview and Setup Requirements

---

## Introduction

This section provides an overview of the main steps involved in creating an Orbix E2A COBOL 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 E2A COBOL application are:

Step	Action
1	<a href="#">“Developing the Application Interfaces” on page 19.</a>
2	<a href="#">“Developing the Server” on page 24.</a>
3	<a href="#">“Developing the Client” on page 35.</a>

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

## The Simple demonstration

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

- An Orbix E2A COBOL server that implements a simple persistent POA-based server.
- An Orbix E2A COBOL 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 E2A Application Server Platform Mainframe Edition, because the client and server are written in the same language and running on the same platform.



---

**The demonstration server**

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

See [“Location of supplied code and JCL” on page 15](#) for details of where you can find an example of the supplied server. See [“Developing the Server” on page 24](#) 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” on page 15](#) for details of where you can find an example of the supplied client. See [“Developing the Client” on page 35](#) for more details of how to develop the client.

---

**Location of supplied code and JCL**

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

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

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

Location	Description
<code>orbixhlq.DEMOS.IDL(SIMPLE)</code>	This is the supplied IDL.
<code>orbixhlq.DEMOS.COBOL.SRC(SIMPLESV)</code>	This is the source code for the batch server mainline program.
<code>orbixhlq.DEMOS.COBOL.SRC(SIMPLES)</code>	This is the source code for the batch server implementation program.
<code>orbixhlq.DEMOS.COBOL.SRC(SIMPLECL)</code>	This is the source code for the client program.
<code>orbixhlq.JCL(LOCATOR)</code>	This JCL runs the Orbix E2A locator daemon.
<code>orbixhlq.JCL(NODEDAEM)</code>	This JCL runs the Orbix E2A node daemon.

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

Location	Description
<code>orbixhlq.DEMOS.COBOL.BUILD.JCL(SIMPLIDL)</code>	This JCL runs the Orbix E2A IDL compiler, to generate COBOL source and copybooks for the batch server. The <code>-s</code> and <code>-z</code> compiler arguments, which generate server mainline and server implementation code respectively, are disabled by default in this JCL.
<code>orbixhlq.DEMOS.COBOL.BUILD.JCL(SIMPLECB)</code>	This JCL compiles the client program.
<code>orbixhlq.DEMOS.COBOL.BUILD.JCL(SIMPLESB)</code>	This JCL compiles and links the batch server mainline and batch server implementation programs.
<code>orbixhlq.DEMOS.COBOL.RUN.JCL(SIMPLESV)</code>	This JCL runs the server.
<code>orbixhlq.DEMOS.COBOL.BUILD.JCL(SIMPLECL)</code>	This JCL runs the client.

**Note:** Other code elements and JCL components are provided for the IMS and CICS versions of the `SIMPLE` demonstration. See [“Getting Started in IMS” on page 49](#) and [“Getting Started in CICS” on page 83](#) for more details of these.

## Supplied copybooks

[Table 2](#) provides a summary of the various copybooks supplied with your product installation that are relevant to batch. In [Table 2](#), *servers* means batch servers, and *clients* means batch clients. Again, `orbixhlq` represents your installation's high-level qualifier.

**Table 2:** *Supplied Copybooks (Sheet 1 of 2)*

Location	Description
<code>orbixhlq.INCLUDE.COPYLIB(CHKERRS)</code>	This contains a COBOL function that can be called both by clients and servers to check if an exception has occurred, and to report that exception.
<code>orbixhlq.INCLUDE.COPYLIB(CORBA)</code>	This is used both by clients and servers. It contains various Orbix E2A COBOL definitions, such as <code>REQUEST-INFO</code> used by the <code>COAREQ</code> function, and <code>ORBIX-STATUS-INFORMATION</code> which is used to register and report exceptions raised by the COBOL runtime.

**Table 2:** *Supplied Copybooks (Sheet 2 of 2)*

Location	Description
<code>orbixhlq.INCLUDE.COPYLIB(CORBATYP)</code>	This is used both by clients and servers. It contains the COBOL typecode representation for IDL basic types.
<code>orbixhlq.INCLUDE.COPYLIB(IORSLCT)</code>	This is used both by clients and servers. It contains the COBOL <code>SELECT</code> statement entry for file processing, for use with the <code>COPY...REPLACING</code> statement.
<code>orbixhlq.INCLUDE.COPYLIB(IORFD)</code>	This is used both by clients and servers. It contains the COBOL <code>FD</code> statement entry for file processing, for use with the <code>COPY...REPLACING</code> statement.
<code>orbixhlq.INCLUDE.COPYLIB(CHKFILE)</code>	This is used both by clients and servers. It is used for file handling error checking.
<code>orbixhlq.INCLUDE.COPYLIB(ROCPARM)</code>	This is used both by clients and servers. It contains the appropriate definitions for a COBOL program to accept parameters from the JCL for use with the <code>ORBARGS</code> API (that is, the <code>argument-string</code> parameter).
<code>orbixhlq.DEMOS.COBOL.COPYLIB</code>	This PDS is used to store all batch copybooks generated by the Orbix E2A IDL compiler when you run the supplied <code>SIMPLIDL</code> JCL for the batch demonstration. It also contains copybooks with Working Storage data definitions and Procedure Division paragraphs for use with the bank, naming, and nested sequences demonstrations.

**Note:** Some other copybooks are provided specifically for use with IMS. See [“Getting Started in IMS” on page 49](#) for more details of these.

---

**Checking JCL components**

When creating the `SIMPLE` application, check that each step involved within the separate JCL components completes with a condition code of zero. If the condition codes are not zero, 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 COBOL source and copybooks from IDL interfaces, and provides a description of the members generated from the supplied `SimpleObject` interface.

---

## Steps to develop application interfaces

The steps to develop the interfaces to your application are:

Step	Action
1	Define public IDL interfaces to the objects required in your system. See <a href="#">“Defining IDL Interfaces” on page 20</a> .
2	Use the Orbix E2A IDL compiler to generate COBOL source code and copybooks from the defined IDL. See <a href="#">“Generating COBOL Source and Copybooks” on page 21</a> .

---

## Defining IDL Interfaces

### Defining the IDL

The first step in writing an Orbix E2A program is to define the IDL interfaces for the objects required in your system. The following is an example of the IDL for the `SimpleObject` interface that is supplied in

`orbixhlq.DEMOS.IDL(SIMPLE)`:

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

### Explanation of the IDL

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

### How the demonstration uses this IDL

For the purposes of this example, the `SimpleObject` CORBA object is implemented in COBOL in the supplied `SIMPLES` 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 interoperable object reference (IOR) from the relevant PDS member. The client invokes the `call_me()` operation on the `SimpleObject` object, and then exits.

---

## Generating COBOL Source and Copybooks

---

### The Orbix E2A IDL compiler

You can use the Orbix E2A IDL compiler to generate COBOL source and copybooks from IDL definitions.

### Orbix E2A IDL compiler configuration

The Orbix E2A IDL compiler uses the Orbix E2A configuration member for its settings. The `SIMPLIDL` JCL that runs the compiler uses a configuration member provided in `orbixhlg.CONFIG(IDL)`. See [“Orbix E2A IDL Compiler” on page 237](#) for more details.

### Running the Orbix E2A IDL compiler

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

```
orbixhlg.DEMOS.COBOL.BUILD.JCL(SIMPLIDL)
```

### Generated source code members

[Table 3](#) shows the server source code members that the Orbix E2A IDL compiler generates, based on the defined IDL.

**Table 3:** *Generated Server Source Code Members*

Member	JCL Keyword Parameter	Description
<i>idlmembernameS</i>	IMPL	This is the server implementation source code member. It contains stub paragraphs for all the callable operations.  The is only generated if you specify the <code>-z</code> argument with the IDL compiler.
<i>idlmembernameSV</i>	IMPL	This is server mainline source code member.  This is only generated if you specify the <code>-s</code> argument with the IDL compiler.

**Note:** For the purposes of this example, the `SIMPLES` server implementation and `SIMPLESV` server mainline are already provided in your product installation. Therefore, the IDL compiler arguments that are used to generate them are not specified in the supplied `SIMPLIDL` JCL. See [“Orbix E2A IDL Compiler” on page 237](#) for more details of the IDL compiler arguments used to generate server source code.

**Generated COBOL copybooks**

[Table 4](#) shows the COBOL copybooks that the Orbix E2A IDL compiler generates, based on the defined IDL.

**Table 4:** *Generated COBOL Copybooks*

Copybook	JCL Keyword Parameter	Description
<code>idlmembername</code>	<code>COPYLIB</code>	This copybook contains data definitions that are used for working with operation parameters and return values for each interface defined in the IDL member.  The name for this copybook does not take a suffix.
<code>idlmembernameX</code>	<code>COPYLIB</code>	This copybook contains data definitions that are used by the COBOL runtime to support the interfaces defined in the IDL member.  This copybook is automatically included in the <code>idlmembername</code> copybook.
<code>idlmembernameD</code>	<code>COPYLIB</code>	This copybook contains procedural code for performing the correct paragraph for the requested operation.  This copybook is automatically included in the <code>idlmembernameS</code> source code member.



## How IDL maps to COBOL copybooks

---

Each IDL interface maps to a group of COBOL data definitions. There is one definition for each IDL operation. A definition contains each of the parameters for the relevant IDL operation in their corresponding COBOL representation. See [“IDL-to-COBOL Mapping” on page 161](#) for details of how IDL types map to COBOL.

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

---

## Member name restrictions

Generated source code member and copybook names are based on the IDL member name. If the IDL member name exceeds six characters, the Orbix E2A IDL compiler uses only the first six characters of the IDL member name when generating the other member names. This allows space for appending the two-character `sv` suffix to the name for the server mainline member, while allowing it to adhere to the eight-character maximum size limit for OS/390 member names. Consequently, all other member names also use only the first six characters of the IDL member name, followed by their individual suffixes, as appropriate.

---

## Location of demonstration copybooks

You can find examples of the copybooks generated for the `SIMPLE` demonstration in the following locations:

- `orbixhlq.DEMOS.COBOL.COPYLIB(SIMPLE)`
- `orbixhlq.DEMOS.COBOL.COPYLIB(SIMPLEX)`
- `orbixhlq.DEMOS.COBOL.COPYLIB(SIMPLED)`

**Note:** These copybooks are not shipped with your product installation. They are generated when you run the supplied `SIMPLIDL` JCL, to run the Orbix E2A IDL compiler.

---

# Developing the Server

---

**Overview**

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

---

**Steps to develop the server**

The steps to develop the server application are:

Step	Action
1	<a href="#">“Writing the Server Implementation” on page 25</a>
2	<a href="#">“Writing the Server Mainline” on page 28</a>
3	<a href="#">“Building the Server” on page 34.</a>

---

# Writing the Server Implementation

---

## The server implementation program

You must implement the server interface by writing a COBOL program that implements each operation in the *idlmembername* copybook. For the purposes of this example, you must write a COBOL program that implements each operation in the *SIMPLE* copybook. When you specify the *-z* argument with the Orbix E2A IDL compiler in this case, it generates a skeleton program called *SIMPLES*, which is a useful starting point.

## Example of the SIMPLES program

The following is an example of the batch *SIMPLES* program:

### Example 1: *The Batch SIMPLES Demonstration (Sheet 1 of 2)*

```
*****
* Identification Division
*****
IDENTIFICATION DIVISION.
PROGRAM-ID.          SIMPLES.

ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY SIMPLE.
COPY CORBA.

01 WS-INTERFACE-NAME          PICTURE X(30).
01 WS-INTERFACE-NAME-LENGTH   PICTURE 9(09) BINARY
                               VALUE 30.

*****
* Procedure Division
*****
PROCEDURE DIVISION.

1      ENTRY "DISPATCH".

2      CALL "COAREQ"          USING REQUEST-INFO.
      SET WS-COAREQ TO TRUE.
      PERFORM CHECK-STATUS.
```

**Example 1:** *The Batch SIMPLES Demonstration (Sheet 2 of 2)*

```

3  * Resolve the pointer reference to the interface name which is
   * the fully scoped interface name
   * Note make sure it can handle the max interface name length
      CALL "STRGET"      USING INTERFACE-NAME
                              WS-INTERFACE-NAME-LENGTH
                              WS-INTERFACE-NAME.

      SET WS-STRGET TO TRUE.
      PERFORM CHECK-STATUS.

*****
* Interface(s)  evaluation:
*****
      MOVE SPACES TO SIMPLE-SIMPLEOBJECT-OPERATION.

      EVALUATE WS-INTERFACE-NAME
      WHEN 'IDL:Simple/SimpleObject:1.0'

4  * Resolve the pointer reference to the operation information
      CALL "STRGET" USING OPERATION-NAME
                              SIMPLE-S-3497-OPERATION-LENGTH
                              SIMPLE-SIMPLEOBJECT-OPERATION

      SET WS-STRGET TO TRUE
      PERFORM CHECK-STATUS
      DISPLAY  "Simple::" SIMPLE-SIMPLEOBJECT-OPERATION
              "invoked"
      END-EVALUATE.

5  COPY SIMPLED.

      GOBACK.

6  DO-SIMPLE-SIMPLEOBJECT-CALL-ME.
      CALL "COAGET"      USING SIMPLE-SIMPLEOBJECT-70FE-ARGS.
      SET WS-COAGET TO TRUE.
      PERFORM CHECK-STATUS.

      CALL "COAPUT"      USING SIMPLE-SIMPLEOBJECT-70FE-ARGS.
      SET WS-COAPUT TO TRUE.
      PERFORM CHECK-STATUS.

*****
* Check Errors Copybook
*****
COPY CHKERRS.

```

## Explanation of the batch SIMPLES program

---

The SIMPLES program can be explained as follows:

1. The DISPATCH logic is automatically coded for you, and the bulk of the code is contained in the SIMPLED copybook. When an incoming request arrives from the network, it is processed by the ORB and a call is made to the DISPATCH entry point.
2. COAREQ is called to provide information about the current invocation request, which is held in the REQUEST-INFO block that is contained in the CORBA copybook.  
COAREQ is called once for each operation invocation—after a request has been dispatched to the server, but before any calls are made to access the parameter values.
3. STRGET is called to copy the characters in the unbounded string pointer for the interface name to the string item representing the fully scoped interface name.
4. STRGET is called again to copy the characters in the unbounded string pointer for the operation name to the string item representing the operation name.
5. The procedural code used to perform the correct paragraph for the requested operation is copied into the program from the SIMPLED copybook.
6. Each operation has skeleton code, with appropriate calls to COAPUT and COAGET to copy values to and from the COBOL structures for that operation's argument list. You must provide a correct implementation for each operation. You must call COAGET and COAPUT, even if your operation takes no parameters and returns no data. You can simply pass in a dummy area as the parameter list.

**Note:** The supplied SIMPLES program is only a suggested way of implementing an interface. It is not necessary to have all operations implemented in the same COBOL program.

## Location of the batch SIMPLES program

---

You can find a complete version of the batch SIMPLES server implementation program in `orbixhlq.DEMOS.COBOL.SRC(SIMPLES)`.

# Writing the Server Mainline

**The server mainline program**

The next step is to write the server mainline program in which to run the server implementation. For the purposes of this example, when you specify the `-s` argument with the Orbix E2A IDL compiler, it generates a program called `SIMPLESV`, which contains the server mainline code.

**Example of the batch SIMPLESV program**

The following is an example of the batch `SIMPLESV` program:

**Example 2:** *The Batch SIMPLESV Demonstration (Sheet 1 of 4)*

```
IDENTIFICATION DIVISION.
PROGRAM-ID.          SIMPLESV.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    COPY IORSLCT REPLACING
        "X-IOR" BY SIMPLE-SIMPLEOBJECT-IOR
        "X-IORFILE" BY "IORFILE"
        "X-IOR-STAT" BY SIMPLE-SIMPLEOBJECT-IOR-STAT.
DATA DIVISION.
FILE SECTION.

    COPY IORFD REPLACING
        "X-IOR" BY SIMPLE-SIMPLEOBJECT-IOR
        "X-REC" BY SIMPLE-SIMPLEOBJECT-REC.

WORKING-STORAGE SECTION.

COPY SIMPLE.
COPY CORBA.

01 ARG-LIST                                PICTURE X(01)
                                           VALUE SPACES.
01 ARG-LIST-LEN                            PICTURE 9(09) BINARY
                                           VALUE 0.
01 ORB-NAME                                PICTURE X(10)
                                           VALUE "simple_orb".
01 ORB-NAME-LEN                            PICTURE 9(09) BINARY
                                           VALUE 10.
01 SERVER-NAME                            PICTURE X(18)
                                           VALUE "simple_persistent ".
```

**Example 2:** *The Batch SIMPLESV Demonstration (Sheet 2 of 4)*

```
01 SERVER-NAME-LEN                PICTURE 9(09) BINARY
                                   VALUE 17.

01 INTERFACE-LIST.
   03 FILLER                      PICTURE X(28)
                                   VALUE "IDL:Simple/SimpleObject:1.0 ".
01 INTERFACE-NAMES-ARRAY REDEFINES INTERFACE-LIST.
   03 INTERFACE-NAME OCCURS 1 TIMES PICTURE X(28).

01 OBJECT-ID-LIST.
   03 FILLER                      PICTURE X(17)
                                   VALUE "my_simple_object ".
01 OBJECT-ID-ARRAY REDEFINES OBJECT-ID-LIST.
   03 OBJECT-IDENTIFIER OCCURS 1 TIMES PICTURE X(17).

01 IOR-REC-LEN                   PICTURE 9(09) BINARY
                                   VALUE 2048.
01 IOR-REC-PTR                   POINTER.
                                   VALUE NULL.

*****
* Status and Obj values for the Interface(s)
*****
01 SIMPLE-SIMPLEOBJECT-IOR-STAT   PICTURE 9(02).
01 SIMPLE-SIMPLEOBJECT-OBJ       POINTER
                                   VALUE NULL.

COPY PROCPARM.

INIT.

1      CALL "ORBSTAT"    USING ORBIX-STATUS-INFORMATION.

      DISPLAY "Initializing the ORB".

2      CALL "ORBARGS"    USING ARG-LIST
                                   ARG-LIST-LEN
                                   ORB-NAME
                                   ORB-NAME-LEN.

      SET WS-ORBARGS TO TRUE.
      PERFORM CHECK-STATUS.

3      CALL "ORBSRVR"    USING SERVER-NAME
                                   SERVER-NAME-LEN.

      SET WS-ORBSRVR TO TRUE.
```

**Example 2:** *The Batch SIMPLESV Demonstration (Sheet 2 of 4)*

```

01 SERVER-NAME-LEN                                PICTURE 9(09) BINARY
                                                    VALUE 17.

01 INTERFACE-LIST.
   03 FILLER                                      PICTURE X(28)
      VALUE "IDL:Simple/SimpleObject:1.0 ".
01 INTERFACE-NAMES-ARRAY REDEFINES INTERFACE-LIST.
   03 INTERFACE-NAME OCCURS 1 TIMES              PICTURE X(28).

01 OBJECT-ID-LIST.
   03 FILLER                                      PICTURE X(17)
      VALUE "my_simple_object ".
01 OBJECT-ID-ARRAY REDEFINES OBJECT-ID-LIST.
   03 OBJECT-IDENTIFIER OCCURS 1 TIMES          PICTURE X(17).

01 IOR-REC-LEN                                    PICTURE 9(09) BINARY
                                                    VALUE 2048.
01 IOR-REC-PTR                                    POINTER.
                                                    VALUE NULL.

*****
* Status and Obj values for the Interface(s)
*****
01 SIMPLE-SIMPLEOBJECT-IOR-STAT                  PICTURE 9(02).
01 SIMPLE-SIMPLEOBJECT-OBJ                      POINTER
                                                    VALUE NULL.

COPY PROCPARM.

INIT.

1      CALL "ORBSTAT"    USING ORBIX-STATUS-INFORMATION.

      DISPLAY "Initializing the ORB".

2      CALL "ORBARGS"    USING ARG-LIST
                                ARG-LIST-LEN
                                ORB-NAME
                                ORB-NAME-LEN.

      SET WS-ORBARGS TO TRUE.
      PERFORM CHECK-STATUS.

3      CALL "ORBSRVR"    USING SERVER-NAME
                                SERVER-NAME-LEN.

      SET WS-ORBSRVR TO TRUE.

```



**Example 2:** *The Batch SIMPLESV Demonstration (Sheet 3 of 4)*

```
PERFORM CHECK-STATUS.

*****
* Interface Section Block
*****

* Generating IOR for interface Simple/SimpleObject
  DISPLAY "Registering the Interface".

4  CALL "ORBREG"      USING SIMPLE-SIMPLEOBJECT-INTERFACE.
   SET WS-ORBREG TO TRUE.

   OPEN OUTPUT SIMPLE-SIMPLEOBJECT-IOR.
   COPY CHKFILE REPLACING
     "X-IOR-STAT" BY SIMPLE-SIMPLEOBJECT-IOR-STAT.

   DISPLAY "Creating the Object".
5  CALL "OBJNEW"      USING SERVER-NAME
                           INTERFACE-NAME
                           OF INTERFACE-NAMES-ARRAY(1)
                           OBJECT-IDENTIFIER
                           OF OBJECT-ID-ARRAY(1)
                           SIMPLE-SIMPLEOBJECT-OBJ.

   SET WS-OBJNEW TO TRUE.
   PERFORM CHECK-STATUS.

6  CALL "OBJTOSTR"    USING SIMPLE-SIMPLEOBJECT-OBJ
                           IOR-REC-PTR.

   SET WS-OBJTOSTR TO TRUE.
   PERFORM CHECK-STATUS.

   CALL "STRGET"      USING IOR-REC-PTR
                           IOR-REC-LEN
                           SIMPLE-SIMPLEOBJECT-REC.

   SET WS-STRGET TO TRUE.
   PERFORM CHECK-STATUS.

   CALL "STRFREE"     USING IOR-REC-PTR.
   SET WS-STRFREE TO TRUE.
   PERFORM CHECK-STATUS.

   DISPLAY "Writing object reference to file".

   WRITE SIMPLE-SIMPLEOBJECT-REC.
```

**Example 2:** *The Batch SIMPLESV Demonstration (Sheet 4 of 4)*

7

```

COPY CHKFILE REPLACING
  "X-IOR-STAT" BY SIMPLE-SIMPLEOBJECT-IOR-STAT.

CLOSE SIMPLE-SIMPLEOBJECT-IOR.
COPY CHKFILE REPLACING
  "X-IOR-STAT" BY SIMPLE-SIMPLEOBJECT-IOR-STAT.

DISPLAY "Giving control to the ORB to process Requests".
CALL "COARUN".
SET WS-COARUN TO TRUE.
PERFORM CHECK-STATUS.

CALL "OBJREL" USING SIMPLE-SIMPLEOBJECT-OBJ.
SET WS-OBJREL TO TRUE.
PERFORM CHECK-STATUS.

EXIT-PRG.
  STOP RUN.

*****
* Check Errors Copybook
*****
COPY CHKERRS.

```

**Explanation of the batch  
SIMPLESV program**

The SIMPLESV program can be explained as follows:

1. ORBSTAT is called to register the ORBIX-STATUS-INFORMATION block that is contained in the CORBA copybook. Registering the ORBIX-STATUS-INFORMATION block allows the COBOL runtime to populate it with exception information, if necessary.
2. ORBARGS is called to initialize a connection to the ORB.
3. ORBSRV is called to set the server name.
4. ORBREG is called to register the IDL interface, SimpleObject, with the Orbix COBOL runtime.
5. OBJNEW is called to create a persistent server object of the SimpleObject type, with an object ID of my\_simple\_object.
6. OBJTOSTR 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. COARUN is called, to enter the ORB::run loop, to allow the ORB to receive and process client requests.

---

# Building the Server

---

Location of the JCL	Sample JCL used to compile and link the batch server mainline and server implementation is in <code>orbixhlq.DEMOS.COBOL.BUILD.JCL(SIMPLESB)</code> .
Resulting load module	When this JCL has successfully executed, it results in a load module that is contained in <code>orbixhlq.DEMOS.COBOL.LOAD(SIMPLESV)</code> .

---

# Developing the Client

---

## Overview

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

**Note:** The Orbix E2A IDL compiler does not generate COBOL client stub code.

## Steps to develop the client

The steps to develop the client application are:

Step	Action
1	<a href="#">"Writing the Client" on page 36.</a>
2	<a href="#">"Building the Client" on page 41.</a>

---

# Writing the Client

## The client program

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

## Example of the `SIMPLECL` program

The following is an example of the `SIMPLECL` program:

### Example 3: *The SIMPLECL Demonstration Program (Sheet 1 of 3)*

```
IDENTIFICATION DIVISION.  
PROGRAM-ID.                                SIMPLECL.  
  
ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.  
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
    COPY IORSLCT REPLACING  
        "X-IOR" BY SIMPLE-SIMPLEOBJECT-IOR  
        "X-IORFILE" BY "IORFILE"  
        "X-IOR-STAT" BY SIMPLE-SIMPLEOBJECT-IOR-STAT.  
DATA DIVISION.  
FILE SECTION.  
  
    COPY IORFD REPLACING  
        "X-IOR" BY SIMPLE-SIMPLEOBJECT-IOR  
        "X-REC" BY SIMPLE-SIMPLEOBJECT-REC.  
  
WORKING-STORAGE SECTION.  
  
COPY SIMPLE.  
COPY CORBA.  
  
01 WS-SIMPLE-IOR                                PICTURE X(2048).  
01 SIMPLE-IOR-LENGTH                            PICTURE 9(9) BINARY  
                                                VALUE 2048.  
01 SIMPLE-SIMPLEOBJECT-IOR-STAT                PICTURE 9(02).  
01 SIMPLE-SIMPLEOBJECT-OBJ                    POINTER  
                                                VALUE NULL.  
01 ARG-LIST                                    PICTURE X(1)  
                                                VALUE SPACES.  
01 ARG-LIST-LEN                                PICTURE 9(09) BINARY  
                                                VALUE 0.
```

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

```
01 ORB-NAME                PICTURE X(10)
                           VALUE "simple_orb".
01 ORB-NAME-LEN            PICTURE 9(09) BINARY
                           VALUE 10.
01 IOR-REC-PTR             POINTER
                           VALUE NULL.
01 IOR-REC-LEN            PICTURE 9(09) BINARY
                           VALUE 2048.

COPY PROCPARM.
1  CALL "ORBSTAT" USING ORBIX-STATUS-INFORMATION.

* ORB initialization
  DISPLAY "Initializing the ORB".
2  CALL "ORBARGS"  USING ARG-LIST
                           ARG-LIST-LEN
                           ORB-NAME
                           ORB-NAME-LEN.

  SET WS-ORBARGS TO TRUE.
  PERFORM CHECK-STATUS.

* Register interface TypeTest
  DISPLAY "Registering the Interface".
3  CALL "ORBREG"  USING SIMPLE-SIMPLEOBJECT-INTERFACE.
  SET WS-ORBREG TO TRUE.
  PERFORM CHECK-STATUS.

*
4  ** Read in the IOR from a file which has been populated
  ** by the server program.
  *

  OPEN INPUT SIMPLE-SIMPLEOBJECT-IOR.
  COPY CHKFILE REPLACING
    "X-IOR-STAT" BY SIMPLE-SIMPLEOBJECT-IOR-STAT.

  DISPLAY "Reading object reference from file".
  READ SIMPLE-SIMPLEOBJECT-IOR.
  COPY CHKFILE REPLACING
    "X-IOR-STAT" BY SIMPLE-SIMPLEOBJECT-IOR-STAT.

  MOVE SIMPLE-SIMPLEOBJECT-REC TO WS-SIMPLE-IOR.

* IOR Record read successfully
  CLOSE SIMPLE-SIMPLEOBJECT-IOR.
  COPY CHKFILE REPLACING
```

**Example 3:** *The SIMPLECL Demonstration Program (Sheet 3 of 3)*

```

      "X-IOR-STAT" BY SIMPLE-SIMPLEOBJECT-IOR-STAT.
5  * Set the COBOL pointer to point to the IOR string
      CALL "STRSET"      USING IOR-REC-PTR
                                IOR-REC-LEN
                                WS-SIMPLE-IOR.

      SET WS-STRSET TO TRUE.
      PERFORM CHECK-STATUS.

6  * Obtain object reference from the IOR
      CALL "STRTOOBJ" USING IOR-REC-PTR
                                SIMPLE-SIMPLEOBJECT-OBJ

      SET WS-STRTOOBJ TO TRUE.
      PERFORM CHECK-STATUS.

      * Releasing the memory
      CALL "STRFREE" USING IOR-REC-PTR.
      SET WS-STRFREE TO TRUE.
      PERFORM CHECK-STATUS.

      SET SIMPLE-SIMPLEOBJECT-CALL-ME TO TRUE
      DISPLAY "invoking Simple::" SIMPLE-SIMPLEOBJECT-OPERATION.

7  CALL "ORBEXEC"  USING SIMPLE-SIMPLEOBJECT-OBJ
                                SIMPLE-SIMPLEOBJECT-OPERATION
                                SIMPLE-SIMPLEOBJECT-70FE-ARGS
                                SIMPLE-USER-EXCEPTIONS.

      SET WS-ORBEXEC TO TRUE.
      PERFORM CHECK-STATUS.

      CALL "OBJREL" USING SIMPLE-SIMPLEOBJECT-OBJ.
      SET WS-OBJREL TO TRUE.
      PERFORM CHECK-STATUS.

      DISPLAY "Simple demo complete.".

EXIT-PRG.
*=====
STOP RUN.

*****
* Check Errors Copybook
*****
      COPY CHKERRS.

```



## Explanation of the SIMPLECL program

---

The SIMPLECL program can be explained as follows:

1. ORBSTAT is called to register the ORBIX-STATUS-INFORMATION block that is contained in the CORBA copybook. Registering the ORBIX-STATUS-INFORMATION block allows the COBOL runtime to populate it with exception information, if necessary.  
You can use the ORBIX-STATUS-INFORMATION data item (in the CORBA copybook) to check the status of any Orbix call. The EXCEPTION-NUMBER numeric data item is important in this case. If this item is 0, it means the call was successful. Otherwise, EXCEPTION-NUMBER holds the system exception number that occurred. You should test this data item after any Orbix call.
2. ORBARGS is called to initialize a connection to the ORB.
3. ORBREG is called to register the IDL interface with the Orbix COBOL runtime.
4. The client reads the stringified object reference for the object from the PDS member that has been populated by the server. For the purposes of this example, the IOR member is contained in `orbixhlq.DEMOS.IORS(SIMPLE)`.
5. STRSET is called to create an unbounded string to which the stringified object reference is copied.
6. STRTOOBJ is called to create an object reference to the server object that is represented by the IOR. This must be done to allow operation invocations on the server. The STRTOOBJ 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.
7. After the object reference is created, ORBEXEC 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. The operation name must have at least one trailing space. The generated operation condition names found in the SIMPLE copybook already handle this.

The same argument description is used by the server, and is found in the `SIMPLE` copybook. For example, See  
`orbixhlq.DEMOS.COBOL.COPYLIB(SIMPLE).`

---

### **Location of the SIMPLECL program**

You can find a complete version of the `SIMPLECL` client program in  
`orbixhlq.DEMOS.COBOL.SRC(SIMPLECL).`

---

## Building the Client

---

### Location of the JCL

Sample JCL used to compile and link the client can be found in the third step of `orbixhlq.DEMOS.COBOL.BUILD.JCL(SIMPLECB)`.

---

### Resulting load module

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

---

# Running the Application

---

## Introduction

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

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

## Steps to run the application

The steps to run the application are:

Step	Action
1	<a href="#">“Starting the Orbix E2A Locator Daemon” on page 43</a> (if it has not already been started).
2	<a href="#">“Starting the Orbix E2A Node Daemon” on page 44</a> (if it has not already been started).
3	<a href="#">“Running the Server and Client” on page 45</a> .

---

## Starting the Orbix E2A Locator Daemon

---

### Overview

An Orbix E2A locator daemon must be running on the server's location domain before you try to run your application. The Orbix E2A 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 E2A 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 E2A locator daemon

If the Orbix E2A locator daemon is not already running, you can use the JCL in `orbixhlq.JCL(LOCATOR)` to start it.

---

### Locator daemon configuration

The Orbix E2A locator daemon uses the Orbix E2A configuration member for its settings. The JCL that you use to start the locator daemon uses a sample configuration member that is provided in `orbixhlq.DOMAINS(FILEDOMA)`.

---

## Starting the Orbix E2A Node Daemon

---

### Overview

An Orbix E2A 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 E2A 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 E2A node daemon

If the Orbix E2A node daemon is not already running, you can use the JCL in `orbixhlq.JCL(NODEDAEM)` to start it.

---

### Node daemon configuration

The Orbix E2A node daemon uses the Orbix E2A configuration member for its settings. The JCL that you use to start the node daemon uses a configuration member that is provided in `orbixhlq.DOMAINS(FILEDOMA)`.

---

## Running the Server and Client

---

### Overview

This section describes how to run the `SIMPLE` demonstration.

---

### JCL to run the server

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

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

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

---

### IOR member for the server

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

---

### JCL to run the client

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

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

---

## Application Output

---

### Server output

The following is an example of the output produced by the server for the `SIMPLE` demonstration:

```
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
```

**Note:** All but the last line of the preceding server output is produced by the `SIMPLESV` server mainline program. The final line is produced by the `SIMPLES` server implementation program.

---

### Client output

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

```
Initializing the ORB
Registering the Interface
Reading object reference from file
invoking Simple::call_me
Simple demo complete.
```

---

### Result

If you receive the preceding client and server output, it means you have successfully created an Orbix E2A COBOL client-server batch application.



# Application Address Space Layout

## Overview

Figure 3 is a graphical overview of the address space layout for an Orbix E2A COBOL application running in batch in a native OS/390 environment. This is shown for the purposes of example and is not meant to reflect a real-world scenario requiring the Orbix E2A Application Server Platform Mainframe Edition.

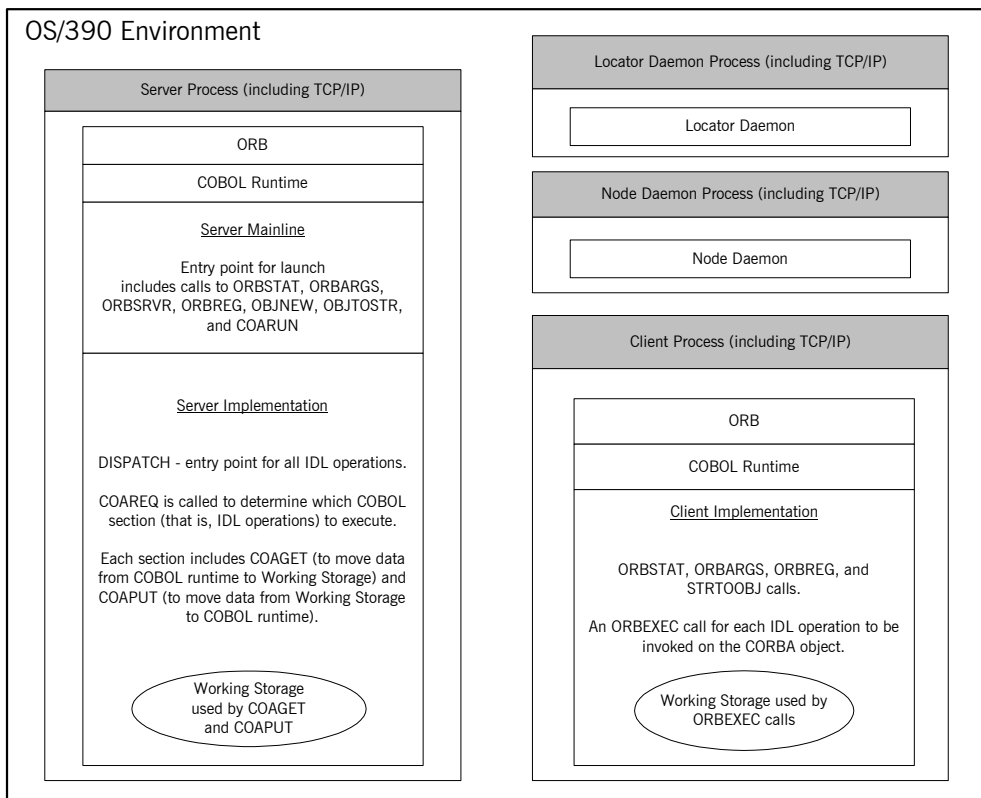


Figure 3: Address Space Layout for an Orbix E2A COBOL Application

**Explanation of the batch server process**


---

The server-side ORB, COBOL runtime, server mainline (launch entry point) and server implementation (`DISPATCH` entry point) are linked into a single load module referred to as the "server". The COBOL runtime marshals data to and from the server implementation working storage, which means there is language-specific translation between C++ and COBOL.

The server runs within its own address space. Link the code as `STATIC` and `NOREENTRANT` (that is, not re-entrant).

The server 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 batch SIMPLESV program” on page 32](#) and [“API Reference” on page 263](#).
  - The APIs called by the server implementation, see [“Explanation of the batch SIMPLES program” on page 27](#) and [“API Reference” on page 263](#).
- 

**Explanation of the daemon processes**

The locator daemon and node daemon each runs in its own address space. See [“Location Domains” on page 10](#) 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 batch client process**

The client-side ORB, COBOL 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 SIMPLECL program” on page 39](#) and [“API Reference” on page 263](#).

# Getting Started in IMS

*This chapter introduces IMS application programming with Orbix E2A, by showing how to use Orbix E2A to develop a simple distributed application that features a COBOL client running in batch and a COBOL server running in IMS.*

## In this chapter

This chapter discusses the following topics:

<a href="#">Overview and Setup Requirements</a>	<a href="#">page 50</a>
<a href="#">Developing the Application Interfaces</a>	<a href="#">page 56</a>
<a href="#">Developing the Server</a>	<a href="#">page 62</a>
<a href="#">Developing and Running the Client</a>	<a href="#">page 76</a>

**Note:** The example provided in this chapter requires use of the IMS adapter, which is supplied as part of the Orbix E2A Application Server Platform Mainframe Edition. See the *IMS Adapter Administrator's Guide* for more details about the IMS adapter.

---

# Overview and Setup Requirements

---

## Overview

This section provides an overview of the main steps involved in creating an Orbix E2A COBOL 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 E2A COBOL application are:

Step	Action
1	<a href="#">“Developing the Application Interfaces” on page 56.</a>
2	<a href="#">“Developing the Server” on page 62.</a>
3	<a href="#">“Developing and Running the Client” on page 76.</a>

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

## The Simple demonstration

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

- An Orbix E2A COBOL server that implements a simple persistent POA-based server.
- An Orbix E2A COBOL 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 E2A Application Server Platform Mainframe Edition, because the client and server are written in the same language and running on the same platform.

---

**The demonstration server**

The server runs in an IMS region. It accepts and processes requests from the client across the network, and communicates with the client via the IMS adapter.

See [“Location of supplied code and JCL” on page 51](#) for details of where you can find an example of the supplied server. See [“Developing the Server” on page 62](#) for more details of how to develop the server.

---

**The demonstration client**

The client runs as a batch job. It 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 via the IMS adapter. When the operation has completed, a reply message is sent back to the client via the IMS adapter. This completes a single remote CORBA invocation.

See [“Location of supplied code and JCL” on page 51](#) for details of where you can find an example of the supplied client. See [“Developing and Running the Client” on page 76](#) 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 IMS `SIMPLE` demonstration 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 the code elements and JCL components that are relevant to the IMS `SIMPLE` demonstration (where `orbixhlq` represents your installation's high-level qualifier).

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

Location	Description
<code>orbixhlq.DEMOS.IDL(SIMPLE)</code>	This is the supplied IDL.
<code>orbixhlq.DEMOS.IMS.COBOI.SRC(SIMPLESV)</code>	This is the source code for the IMS server mainline program, which is generated when you run the JCL in <code>orbixhlq.DEMOS.IMS.COBOI.BUILD.JCL(SIMPLIDL)</code> . (The IMS server mainline code is not shipped with the product. You must run the <code>SIMPLIDL</code> JCL to generate it.)
<code>orbixhlq.DEMOS.IMS.COBOI.SRC(SIMPLES)</code>	This is the source code for the IMS server implementation program.

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

Location	Description
<code>orbixhlq.DEMOS.COBOL.SRC(SIMPLECL)</code>	This is the source code for the batch client program.
<code>orbixhlq.DEMOS.IMS.COBOL.BUILD.JCL(SIMPLIDL)</code>	<p>This JCL runs the Orbix E2A IDL compiler, to generate COBOL source and copybooks for the IMS server. This JCL specifies the <code>-s</code> and <code>-TIMS</code> compiler arguments, to generate IMS server mainline code. It does not specify the <code>-z</code> argument, which generates server implementation code.</p> <p>This JCL also specifies the <code>-mfa</code> and <code>-ttransaction_name</code> arguments, to generate the adapter mapping file, which is then written to <code>orbixhlq.DEMOS.IMS.MFAMAP(SIMPLEA)</code>. The contents of the <code>SIMPLEA</code> member are <code>(Simple/SimpleObject,call_me,SIMPLESV)</code> (that is, fully qualified interface name followed by operation name followed by IMS transaction name). See the <i>IMS Adapter Administrator's Guide</i> for more details about generating adapter mapping files.</p>
<code>orbixhlq.DEMOS.COBOL.BUILD.JCL(SIMPLECB)</code>	This JCL compiles the client program.
<code>orbixhlq.DEMOS.IMS.COBOL.BUILD.JCL(SIMPLESB)</code>	This JCL compiles and links the IMS server mainline and IMS server implementation programs.
<code>orbixhlq.DEMOS.IMS.COBOL.BUILD.JCL(SIMPLREG)</code>	This JCL registers the IDL in the Interface Repository.
<code>orbixhlq.DEMOS.IMS.COBOL.BUILD.JCL(SIMPLIOR)</code>	This JCL obtains the IOR that the client of the IMS server requires to locate the IMS adapter.
<code>orbixhlq.DEMOS.COBOL.RUN.JCL(SIMPLECL)</code>	This JCL runs the batch client.

**Note:** Other code elements and JCL components are provided for the batch and CICS versions of the `SIMPLE` demonstration. See [“Getting Started in Batch” on page 13](#) and [“Getting Started in CICS” on page 83](#) for more details of these.

## Supplied copybooks

Table 6 provides a summary of the various copybooks supplied with your product installation that are relevant to IMS. In Table 6, *servers* means IMS servers, and *clients* means batch clients. Again, *orbixhlq* represents your installation's high-level qualifier.

**Table 6:** *Supplied Copybooks (Sheet 1 of 3)*

Location	Description
<code>orbixhlq.INCLUDE.COPYLIB(CHKERRS)</code>	This is used by clients. It is not used by IMS servers. It contains a COBOL function that can be called by the client, to check if an exception has occurred, and to report that exception.
<code>orbixhlq.INCLUDE.COPYLIB(CERRSMFA)</code>	This is used by servers. It contains a COBOL function that can be called by the IMS server, to check if an exception has occurred, and to report that exception.
<code>orbixhlq.INCLUDE.COPYLIB(CORBA)</code>	This is used both by clients and servers. It contains various Orbix E2A COBOL definitions, such as <code>REQUEST-INFO</code> used by the <code>COAREQ</code> function, and <code>ORBIX-STATUS-INFORMATION</code> which is used to register and report exceptions raised by the COBOL runtime.
<code>orbixhlq.INCLUDE.COPYLIB(CORBATYP)</code>	This is used both by clients and servers. It contains the COBOL typecode representation for IDL basic types.
<code>orbixhlq.INCLUDE.COPYLIB(IORSLCT)</code>	This is used by clients. It is not used by IMS servers. It contains the COBOL <code>SELECT</code> statement entry for file processing, for use with the <code>COPY...REPLACING</code> statement.
<code>orbixhlq.INCLUDE.COPYLIB(IORFD)</code>	This is used by clients. It is not used by IMS servers. It contains the COBOL <code>FD</code> statement entry for file processing, for use with the <code>COPY...REPLACING</code> statement.
<code>orbixhlq.INCLUDE.COPYLIB(CHKFILE)</code>	This is used by clients. It is not used by IMS servers. It is used for file handling error checking.

**Table 6:** *Supplied Copybooks (Sheet 2 of 3)*

Location	Description
<code>orbixhlq.INCLUDE.COPYLIB(PROCPARM)</code>	This is used by clients. It is not used by IMS servers. It contains the appropriate definitions for a COBOL program to accept parameters from the JCL for use with the ORBARGS API (that is, the <code>argument-string</code> parameter).
<code>orbixhlq.DEMOS.IMS.COBOLE.COPYLIB</code>	This PDS is used to store all IMS copybooks that are generated by the Orbix E2A IDL compiler when you run the supplied <code>SIMPLIDL</code> JCL for the IMS demonstration. It also contains copybooks with Working Storage data definitions and Procedure Division paragraphs for the nested sequences demonstration.
<code>orbixhlq.DEMOS.IMS.MFAMAP</code>	This PDS is empty at installation time. It is used to store the IMS adapter mapping member that is generated by the Orbix E2A IDL compiler when you run the supplied <code>SIMPLIDL</code> JCL.
<code>orbixhlq.INCLUDE.COPYLIB(LSIMSPCB)</code>	This is specific to IMS and is used in IMS server mainline programs. It contains the linkage section definitions of the program communication blocks (PCBs).
<code>orbixhlq.INCLUDE.COPYLIB(UPDTPCBS)</code>	This is specific to IMS and is used in IMS server mainline programs. It contains a paragraph that is used to move the PCB data defined in the linkage section (in the <code>LSIMSPCB</code> copybook) to the corresponding working storage defined data (in the <code>WSIMSPCB</code> copybook).



Table 6: Supplied Copybooks (Sheet 3 of 3)

Location	Description
<code>orbixhlq.INCLUDE.COPYLIB(WSIMSPCB)</code>	<p>This is specific to IMS and is used in IMS server mainline and implementation programs. It contains the working storage definitions of the PCB data. The server mainline uses the paragraph defined in the <code>UPDTPCBS</code> copybook, to populate the <code>WSIMSPCB</code> copybook with PCB data from the <code>LSIMSPCB</code> copybook. This allows the server implementation to access the PCB data, if required.</p> <p><b>Note:</b> This data is populated in the supplied demonstrations, but it is not used.</p>

Checking JCL components

When creating the `SIMPLE` application, check that each step involved within the separate JCL components completes with a condition code of zero. If the condition codes are not zero, 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 COBOL source and copybooks from IDL interfaces, and provides a description of the members generated from the supplied `SimpleObject` interface.

## Steps to develop application interfaces

The steps to develop the interfaces to your application are:

Step	Action
1	Define public IDL interfaces to the objects required in your system. See <a href="#">“Defining IDL Interfaces” on page 57</a> .
2	Use the Orbix E2A IDL compiler to generate COBOL source code and copybooks from the defined IDL. See <a href="#">“Generating COBOL Source and Copybooks” on page 58</a> .

---

## Defining IDL Interfaces

### Defining the IDL

The first step in writing an Orbix E2A program is to define the IDL interfaces for the objects required in your system. The following is an example of the IDL for the `SimpleObject` interface that is supplied in

`orbixhlq.DEMOS.IDL(SIMPLE)`:

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

### Explanation of the IDL

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

### How the demonstration uses this IDL

For the purposes of this example, the `SimpleObject` CORBA object is implemented in COBOL in the supplied `SIMPLES` 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 interoperable object reference (IOR) from the relevant PDS member. The client invokes the `call_me()` operation on the `SimpleObject` object, and then exits.

---

# Generating COBOL Source and Copybooks

**The Orbix E2A IDL compiler**

You can use the Orbix E2A IDL compiler to generate COBOL source and copybooks from IDL definitions.

**Orbix E2A IDL compiler configuration**

The Orbix E2A IDL compiler uses the Orbix E2A configuration member for its settings. The `SIMPLIDL` JCL that runs the compiler uses a configuration member provided in `orbixhlq.CONFIG(IDL)`. See [“Orbix E2A IDL Compiler”](#) in [Chapter 6](#) for more information. The COBOL source for the compiler is generated from the following job:

## Generated source code members

[Table 7](#) shows the server source code members that the Orbix E2A IDL compiler generates, based on the defined IDL.

**Table 7:** *Generated Server Source Code Members*

Member	JCL Keyword Parameter	Description
<i>idlmembernameS</i>	IMPL	<p>This is the IMS server implementation source code member. It contains stub paragraphs for all the callable operations.</p> <p>This is only generated if you specify both the <code>-z</code> and <code>-TIMS</code> arguments with the IDL compiler.</p>
<i>idlmembernameSV</i>	IMPL	<p>This is the IMS server mainline source code member.</p> <p>This is only generated if you specify both the <code>-s</code> and <code>-TIMS</code> arguments with the IDL compiler.</p>

**Note:** For the purposes of this example, the `SIMPLES` server implementation is already provided in your product installation. Therefore, the `-z` IDL compiler argument used to generate it is not specified in the supplied `SIMPLIDL` JCL. The `SIMPLESV` server mainline is not already provided, so the `-s` argument used to generate it is specified in the supplied JCL. See [“Orbix E2A IDL Compiler” on page 237](#) for more details of the `-s`, `-z`, and `-TIMS` arguments used to generate IMS server code.

Generated COBOL copybooks

Table 8 shows the COBOL copybooks that the Orbix E2A IDL compiler generates, based on the defined IDL.

Table 8: *Generated COBOL Copybooks*

Copybook	JCL Keyword Parameter	Description
<i>idlmembername</i>	COPYLIB	This copybook contains data definitions that are used for working with operation parameters and return values for each interface defined in the IDL member.  The name for this copybook does not take a suffix.
<i>idlmembernameX</i>	COPYLIB	This copybook contains data definitions that are used by the COBOL runtime to support the interfaces defined in the IDL member.  This copybook is automatically included in the <i>idlmembername</i> copybook.
<i>idlmembernameD</i>	COPYLIB	This copybook contains procedural code for performing the correct paragraph for the requested operation.  This copybook is automatically included in the <i>idlmembernameS</i> source code member.

How IDL maps to COBOL copybooks

Each IDL interface maps to a group of COBOL data definitions. There is one definition for each IDL operation. A definition contains each of the parameters for the relevant IDL operation in their corresponding COBOL representation. See [“IDL-to-COBOL Mapping” on page 161](#) for details of how IDL types map to COBOL.

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

Generated adapter mapping member

Table 9 shows the IMS adapter mapping member that the Orbix E2A IDL compiler generates, based on the defined IDL.

Table 9: *Generated IMS Adapter Mapping Member*

Copybook	JCL Keyword Parameter	Description
<i>idlmembernameA</i>	MEMBER	This is a simple text file that determines what interfaces and operations the IMS adapter supports, and the IMS transaction names to which it should map each IDL operation.

Member name restrictions

Generated source code member, copybook, and mapping member names are all based on the IDL member name. If the IDL member name exceeds six characters, the Orbix E2A IDL compiler uses only the first six characters of the IDL member name when generating the other member names. This allows space for appending the two-character `sv` suffix to the name for the server mainline member, while allowing it to adhere to the eight-character maximum size limit for OS/390 member names. Consequently, all other member names also use only the first six characters of the IDL member name, followed by their individual suffixes, as appropriate.

Location of demonstration copybooks and mapping member

You can find examples of the copybooks and IMS adapter mapping member generated for the `SIMPLE` demonstration in the following locations:

- `orbixhlq.DEMOS.IMS.COBOL.COPYLIB(SIMPLE)`
- `orbixhlq.DEMOS.IMS.COBOL.COPYLIB(SIMPLEX)`
- `orbixhlq.DEMOS.IMS.COBOL.COPYLIB(SIMPLED)`
- `orbixhlq.DEMOS.IMS.MFAMAP(SIMPLEA)`

**Note:** These copybooks and mapping member are not shipped with your product installation. They are generated when you run the supplied `SIMPLIDL` JCL, to run the Orbix E2A IDL compiler.

---

# Developing the Server

---

**Overview**

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

---

**Steps to develop the server**

The steps to develop the server application are:

Step	Action
1	<a href="#">“Writing the Server Implementation” on page 63.</a>
2	<a href="#">“Writing the Server Mainline” on page 67.</a>
3	<a href="#">“Building the Server” on page 72.</a>
4	<a href="#">“Preparing the Server to Run in IMS” on page 73.</a>



---

## Writing the Server Implementation

---

### The server implementation program

You must implement the server interface by writing a COBOL program that implements each operation in the *idlmembername* copybook. For the purposes of this example, you must write a COBOL program that implements each operation in the *SIMPLE* copybook. When you specify the *-z* and *-TIMS* arguments with the Orbix E2A IDL compiler in this case, it generates a skeleton program called *SIMPLES*, which is a useful starting point.

---

### Example of the IMS SIMPLES program

The following is an example of the IMS *SIMPLES* program:

#### Example 4: *The IMS SIMPLES Demonstration (Sheet 1 of 2)*

```
*****
* Identification Division
*****
IDENTIFICATION DIVISION.
PROGRAM-ID.                SIMPLES.

ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY SIMPLE.
COPY CORBA.
1 COPY WSIMSPCB.

01 WS-INTERFACE-NAME                PICTURE X(30).
01 WS-INTERFACE-NAME-LENGTH         PICTURE 9(09) BINARY
                                     VALUE 30.

*****
* Procedure Division
*****
PROCEDURE DIVISION.
2 ENTRY "DISPATCH".
3 CALL "COAREQ" USING REQUEST-INFO.
  SET WS-COAREQ TO TRUE.
  PERFORM CHECK-STATUS.
```

**Example 4:** *The IMS SIMPLES Demonstration (Sheet 2 of 2)*

```

4  * Resolve the pointer reference to the interface name which is
   * the fully scoped interface name
   * Note make sure it can handle the max interface name length
      CALL "STRGET"      USING INTERFACE-NAME
                              WS-INTERFACE-NAME-LENGTH
                              WS-INTERFACE-NAME.

      SET WS-STRGET TO TRUE.
      PERFORM CHECK-STATUS.

*****
* Interface(s)  evaluation:
*****
      MOVE SPACES TO SIMPLE-SIMPLEOBJECT-OPERATION.

      EVALUATE WS-INTERFACE-NAME
      WHEN 'IDL:Simple/SimpleObject:1.0'

5  * Resolve the pointer reference to the operation information
      CALL "STRGET" USING OPERATION-NAME
                              SIMPLE-S-3497-OPERATION-LENGTH
                              SIMPLE-SIMPLEOBJECT-OPERATION

      SET WS-STRGET TO TRUE
      PERFORM CHECK-STATUS
      DISPLAY  "Simple::" SIMPLE-SIMPLEOBJECT-OPERATION
              "invoked"
      END-EVALUATE.

6  COPY SIMPLED.

      GOBACK.

7  DO-SIMPLE-SIMPLEOBJECT-CALL-ME.
      CALL "COAGET"      USING SIMPLE-SIMPLEOBJECT-70FE-ARGS.
      SET WS-COAGET TO TRUE.
      PERFORM CHECK-STATUS.

      CALL "COAPUT"      USING SIMPLE-SIMPLEOBJECT-70FE-ARGS.
      SET WS-COAPUT TO TRUE.
      PERFORM CHECK-STATUS.

*****
* Check Errors Copybook
*****

8  COPY CERRSMFA.

```

## Explanation of the IMS SIMPLES program

---

The SIMPLES program can be explained as follows:

1. The `COPY WSIMSPCB` statement provides access to IMS PCBs.
2. The `DISPATCH` logic is automatically coded for you, and the bulk of the code is contained in the `SIMPLED` copybook. When an incoming request arrives from the network, it is processed by the ORB and a call is made to the `DISPATCH` entry point.
3. `COAREQ` is called to provide information about the current invocation request, which is held in the `REQUEST-INFO` block that is contained in the `CORBA` copybook.  
`COAREQ` is called once for each operation invocation—after a request has been dispatched to the server, but before any calls are made to access the parameter values.
4. `STRGET` is called to copy the characters in the unbounded string pointer for the interface name to the string item representing the fully scoped interface name.
5. `STRGET` is called again to copy the characters in the unbounded string pointer for the operation name to the string item representing the operation name.
6. The procedural code used to perform the correct paragraph for the requested operation is copied into the program from the `SIMPLED` copybook.
7. Each operation has skeleton code, with appropriate calls to `COAPUT` and `COAGET` to copy values to and from the COBOL structures for that operation's argument list. You must provide a correct implementation for each operation. You must call `COAGET` and `COAPUT`, even if your operation takes no parameters and returns no data. You can simply pass in a dummy area as the parameter list.
8. The IMS server implementation uses a `COPY CERRSMFA` statement instead of `COPY CHKERRS`.

**Note:** The supplied SIMPLES program is only a suggested way of implementing an interface. It is not necessary to have all operations implemented in the same COBOL program.

## Location of the IMS SIMPLES program

---

You can find a complete version of the IMS `SIMPLES` server implementation program in `orbixhlq.DEMOS.IMS.COBOL.SRC(SIMPLES)`.

---

# Writing the Server Mainline

---

## The server mainline program

The next step is to write the server mainline program in which to run the server implementation. For the purposes of this example, when you specify the `-S` and `-TIMS` arguments with the Orbix E2A IDL compiler, it generates a program called `SIMPLESV`, 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 adapter.

---

## Example of the IMS SIMPLESV program

The following is an example of the IMS `SIMPLESV` program:

### Example 5: *The IMS SIMPLESV Demonstration (Sheet 1 of 3)*

```
IDENTIFICATION DIVISION.  
PROGRAM-ID.          SIMPLESV.  
ENVIRONMENT DIVISION.  
DATA DIVISION.  
  
WORKING-STORAGE SECTION.  
  
COPY SIMPLE.  
COPY CORBA.  
COPY WSIMSPCB.  
  
01 ARG-LIST                                PICTURE X(01)  
                                           VALUE SPACES.  
01 ARG-LIST-LEN                            PICTURE 9(09) BINARY  
                                           VALUE 0.  
01 ORB-NAME                                PICTURE X(10)  
                                           VALUE "simple_orb".  
01 ORB-NAME-LEN                            PICTURE 9(09) BINARY  
                                           VALUE 10.  
  
01 SERVER-NAME                             PICTURE X(07)  
                                           VALUE "simple ".  
01 SERVER-NAME-LEN                         PICTURE 9(09) BINARY  
                                           VALUE 6.
```

**Example 5:** *The IMS SIMPLESV Demonstration (Sheet 2 of 3)*

```

01 INTERFACE-LIST.
03 FILLER                                PICTURE X(28)
   VALUE "IDL:Simple/SimpleObject:1.0 ".
01 INTERFACE-NAMES-ARRAY REDEFINES INTERFACE-LIST.
03 INTERFACE-NAME OCCURS 1 TIMES        PICTURE X(28).

01 OBJECT-ID-LIST.
03 FILLER                                PICTURE X(27)
   VALUE "Simple/SimpleObject_object ".
01 OBJECT-ID-ARRAY REDEFINES OBJECT-ID-LIST.
03 OBJECT-IDENTIFIER OCCURS 1 TIMES     PICTURE X(27).

*****
* Object values for the Interface(s)
*****

01 SIMPLE-SIMPLEOBJECT-OBJ              POINTER
                                         VALUE NULL.

COPY LSIMSPCB.

PROCEDURE DIVISION USING LS-IO-PCB, LS-ALT-PCB.

INIT.
   PERFORM UPDATE-WS-PCBS.

1   CALL "ORBSTAT"    USING ORBIX-STATUS-INFORMATION.
   SET WS-ORBSTAT TO TRUE.
   PERFORM CHECK-STATUS.

2   CALL "ORBARGS"    USING ARG-LIST
                               ARG-LIST-LEN
                               ORB-NAME
                               ORB-NAME-LEN.
   SET WS-ORBARGS TO TRUE.
   PERFORM CHECK-STATUS.

3   CALL "ORBSRVR"    USING SERVER-NAME
                               SERVER-NAME-LEN.
   SET WS-ORBSRVR TO TRUE.
   PERFORM CHECK-STATUS.

*****
* Interface Section Block
*****

```

**Example 5:** *The IMS SIMPLESV Demonstration (Sheet 2 of 3)*

```
01 INTERFACE-LIST.
03 FILLER                                PICTURE X(28)
   VALUE "IDL:Simple/SimpleObject:1.0 ".
01 INTERFACE-NAMES-ARRAY REDEFINES INTERFACE-LIST.
03 INTERFACE-NAME OCCURS 1 TIMES        PICTURE X(28).

01 OBJECT-ID-LIST.
03 FILLER                                PICTURE X(27)
   VALUE "Simple/SimpleObject_object ".
01 OBJECT-ID-ARRAY REDEFINES OBJECT-ID-LIST.
03 OBJECT-IDENTIFIER OCCURS 1 TIMES     PICTURE X(27).

*****
* Object values for the Interface(s)
*****

01 SIMPLE-SIMPLEOBJECT-OBJ                POINTER
                                           VALUE NULL.

COPY LSIMSPCB.

PROCEDURE DIVISION USING LS-IO-PCB, LS-ALT-PCB.

INIT.
   PERFORM UPDATE-WS-PCBS.

1   CALL "ORBSTAT"    USING ORBIX-STATUS-INFORMATION.
   SET WS-ORBSTAT TO TRUE.
   PERFORM CHECK-STATUS.

2   CALL "ORBARGS"    USING ARG-LIST
                               ARG-LIST-LEN
                               ORB-NAME
                               ORB-NAME-LEN.
   SET WS-ORBARGS TO TRUE.
   PERFORM CHECK-STATUS.

3   CALL "ORBSVR"     USING SERVER-NAME
                               SERVER-NAME-LEN.
   SET WS-ORBSVR TO TRUE.
   PERFORM CHECK-STATUS.

*****
* Interface Section Block
*****
```

**Example 5:** *The IMS SIMPLESV Demonstration (Sheet 3 of 3)*

```

*   Generating Object Reference for interface Simple/SimpleObject

4   CALL "ORBREG"  USING SIMPLE-SIMPLEOBJECT-INTERFACE.
    SET WS-ORBREG TO TRUE.
    PERFORM CHECK-STATUS.

5   CALL "OBJNEW"  USING SERVER-NAME
                                INTERFACE-NAME OF INTERFACE-NAMES-ARRAY(1)
                                OBJECT-IDENTIFIER OF OBJECT-ID-ARRAY(1)
                                SIMPLE-SIMPLEOBJECT-OBJ.

    SET WS-OBJNEW TO TRUE.
    PERFORM CHECK-STATUS.

6   CALL "COARUN".
    SET WS-COARUN TO TRUE.
    PERFORM CHECK-STATUS.

EXIT-PRG.
GOBACK.

*****
*   Populate the working storage PCB definitions
*****
COPY UPDTPCBS.

*****
*   Check Errors Copybook
*****
COPY CERRSMFA.

```

**Explanation of the IMS SIMPLESV program**

The SIMPLESV program can be explained as follows:

1. ORBSTAT is called to register the ORBIX-STATUS-INFORMATION block that is contained in the CORBA copybook. Registering the ORBIX-STATUS-INFORMATION block allows the COBOL runtime to populate it with exception information, if necessary.
2. ORBARGS is called to initialize a connection to the ORB.
3. ORBSRV is called to set the server name.



4. ORBREG is called to register the IDL interface, `SimpleObject`, with the Orbix COBOL runtime.
  5. OBJNEW is called to create a persistent server object of the `SimpleObject` type, with an object ID of `my_simple_object`.
  6. COARUN 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 adapter sends to IMS. If the transaction has been defined as WFI, multiple requests can be processed in the `COARUN` loop; otherwise, `COARUN` processes only one request.
- 

#### Location of the IMS SIMPLESV program

You can find a complete version of the IMS `SIMPLESV` server mainline program in `orbixhlq.DEMOS.IMS.COBOL.SRC(SIMPLESV)` after you have run the supplied `SIMPLIDL` JCL to run the Orbix E2A IDL compiler.

---

## Building the Server

---

### Location of the JCL

Sample JCL used to compile and link the IMS server mainline and server implementation is in `orbixhlq.DEMOS.IMS.COBOL.BUILD.JCL(SIMPLESB)`.

---

### Resulting load module

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

---

# Preparing the Server to Run in IMS

**Overview** This section describes the required steps to allow the server to run in an IMS MPP region. When all the steps in this section have been completed, the server is started automatically within IMS, as required.

**Steps** The following steps are required to enable an IMS server to run in an IMS MPP region:

Step	Action
1	Define a transaction definition for IMS.
2	Provide the IMS server load module to an IMS MPP region.
3	Generate mapping file entries for the IMS adapter.
4	Add the IDL to the Interface Repository.
5	Obtain the IOR for use by the client program.

**Step 1—Defining transaction definition for IMS** A transaction definition must be created for the server, to allow it to run in IMS. The following is the transaction definition for the supplied demonstration:

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

**Step 2—Providing load module to IMS region** Ensure that the `orbixhlq.DEMOS.IMS.COBOL.LOAD` PDS is added to the STEPLIB for the IMS region that is to run the transaction, or copy the SIMPLESV load module to a PDS in the STEPLIB of the relevant IMS region.

**Step 3—Generating mapping file entries**

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

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

Adapter mapping file generation is performed as part of the supplied `orbixhlq.DEMOS.IMS.COBOL.BUILD.JCL(SIMPLIDL)` JCL. The `-mfa: -ttransaction_name` argument with the IDL compiler generates the mapping file. For the purposes of this example, `transaction_name` is replaced with `SIMPLESV`. An `IDL MFA DD` statement must also be provided in the JCL, to specify the PDS into which the mapping file is generated. See the *IMS Adapter Administrator's Guide* for full details about adapter mapping files.

**Step 4—Adding IDL to interface repository**

The IMS adapter needs to be able to obtain the IDL for the COBOL server from the Interface Repository, so that it knows what data types it has to marshal into IMS for the server, and what data types it should expect back from the IMS transaction. Ensure that the relevant IDL for the server has been added to the Interface Repository before the IMS adapter is started.

To add IDL to the Interface Repository, the Interface Repository must be running. You can use the JCL in `orbixhlq.JCL(IFR)` to start it. The Interface Repository uses the Orbix E2A configuration member for its settings. The JCL that you use to start the Interface Repository uses a configuration member that is provided in `orbixhlq.DOMAINS(FILEDOMA)`.

The following JCL that adds IDL to the Interface Repository is supplied in `orbixhlq.DEMOS.IMS.COBOL.BUILD.JCL(SIMPLEREG)`.

```
//          JCLLIB ORDER=(HLQ.ASP50.PROCS)
//          INCLUDE MEMBER=(ORXVARS)
//IDLCBL    EXEC ORXIDL,
//          SOURCE=SIMPLE,
//          IDL=&ORBIX..DEMOS.IDL,
//          COPYLIB=&ORBIX..DEMOS.IMS.COBOL.COPYLIB,
//          IMPL=&ORBIX..DEMOS.IMS.COBOL.SRC,
//          IDLPARM=' -R'
```

## Step 5—Obtaining the adapter IOR

The final step is to use the `resolve` command, to obtain the IOR that the client needs to locate the IMS adapter. To obtain the IMS adapter's IOR, the following prerequisites apply:

- The Interface Repository must be running and contain the relevant IDL. See [“Step 4—Adding IDL to interface repository” on page 74](#) for details of how to start it, if it is not already running.
- The IMS adapter must be running. See the *IMS Adapter Administrator's Guide* for more details about it and how to start it.
- The IMS adapter mapping file must contain the relevant mapping entries. For the purposes of this example, ensure that the `orbixhlq.DEMOS.IMS.MFAMAP(SIMPLEA)` mapping member is being used. See the *IMS Adapter Administrator's Guide* for details about adapter mapping files.

The following JCL, which includes the `resolve` command to obtain the IOR, is supplied in `orbixhlq.DEMOS.IMS.COBOL.BUILD.JCL(SIMPLIOR)`.

```
//          JCLLIB ORDER=(HLQ.ASP50.PROCS)
//          INCLUDE MEMBER=(ORXVARS)
//REG      EXEC PROC=ORXADMIN,
//  PPARM='mfa resolve Simple/SimpleObject > DD:IOR'
//IOR DD DSN=&ORBIX..DEMOS.IORS(SIMPLE),DISP=SHR
```

---

# Developing and Running the Client

---

## Overview

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

**Note:** The Orbix E2A IDL compiler does not generate COBOL client stub code.

## Steps to develop the client

The steps to develop and run the client application are:

Step	Action
1	<a href="#">"Writing the Client" on page 77.</a>
2	<a href="#">"Building and Running the Client" on page 82.</a>

---

---

---

**Example 6:** *The SIMPLECL Demonstration Program (Sheet 2 of 3)*

```

01 ORB-NAME                PICTURE X(10)
                           VALUE "simple_orb".
01 ORB-NAME-LEN            PICTURE 9(09) BINARY
                           VALUE 10.
01 IOR-REC-PTR             POINTER
                           VALUE NULL.
01 IOR-REC-LEN             PICTURE 9(09) BINARY
                           VALUE 2048.

COPY PROCPARM.
1   CALL "ORBSTAT" USING ORBIX-STATUS-INFORMATION.

*   ORB initialization
    DISPLAY "Initializing the ORB".
2   CALL "ORBARGS"  USING ARG-LIST
                           ARG-LIST-LEN
                           ORB-NAME
                           ORB-NAME-LEN.

    SET WS-ORBARGS TO TRUE.
    PERFORM CHECK-STATUS.

*   Register interface TypeTest
    DISPLAY "Registering the Interface".
3   CALL "ORBREG"  USING SIMPLE-SIMPLEOBJECT-INTERFACE.
    SET WS-ORBREG TO TRUE.
    PERFORM CHECK-STATUS.

*
4  **   Read in the IOR from a file which has been populated
    **   by the server program.
    *

    OPEN INPUT SIMPLE-SIMPLEOBJECT-IOR.
    COPY CHKFILE REPLACING
        "X-IOR-STAT" BY SIMPLE-SIMPLEOBJECT-IOR-STAT.

    DISPLAY "Reading object reference from file".
    READ SIMPLE-SIMPLEOBJECT-IOR.
    COPY CHKFILE REPLACING
        "X-IOR-STAT" BY SIMPLE-SIMPLEOBJECT-IOR-STAT.

    MOVE SIMPLE-SIMPLEOBJECT-REC TO WS-SIMPLE-IOR.

*   IOR Record read successfully
    CLOSE SIMPLE-SIMPLEOBJECT-IOR.
    COPY CHKFILE REPLACING

```



**Example 6:** *The SIMPLECL Demonstration Program (Sheet 3 of 3)*

```
"X-IOR-STAT" BY SIMPLE-SIMPLEOBJECT-IOR-STAT.

5  * Set the COBOL pointer to point to the IOR string
    CALL "STRSET"      USING IOR-REC-PTR
                                IOR-REC-LEN
                                WS-SIMPLE-IOR.

    SET WS-STRSET TO TRUE.
    PERFORM CHECK-STATUS.

6  * Obtain object reference from the IOR
    CALL "STRTOOBJ"     USING IOR-REC-PTR
                                SIMPLE-SIMPLEOBJECT-OBJ

    SET WS-STRTOOBJ TO TRUE.
    PERFORM CHECK-STATUS.

    * Releasing the memory
    CALL "STRFREE"      USING IOR-REC-PTR.
    SET WS-STRFREE TO TRUE.
    PERFORM CHECK-STATUS.

    SET SIMPLE-SIMPLEOBJECT-CALL-ME TO TRUE
    DISPLAY "invoking Simple::" SIMPLE-SIMPLEOBJECT-OPERATION.

7  CALL "ORBEXEC"       USING SIMPLE-SIMPLEOBJECT-OBJ
                                SIMPLE-SIMPLEOBJECT-OPERATION
                                SIMPLE-SIMPLEOBJECT-70FE-ARGS
                                SIMPLE-USER-EXCEPTIONS.

    SET WS-ORBEXEC TO TRUE.
    PERFORM CHECK-STATUS.

    CALL "OBJREL"       USING SIMPLE-SIMPLEOBJECT-OBJ.
    SET WS-OBJREL TO TRUE.
    PERFORM CHECK-STATUS.

    DISPLAY "Simple demo complete.".

EXIT-PRG.
*=====
STOP RUN.

*****
* Check Errors Copybook
*****
    COPY CHKERRS.
```

## Explanation of the SIMPLECL program

The SIMPLECL program can be explained as follows:

1. ORBSTAT is called to register the ORBIX-STATUS-INFORMATION block that is contained in the CORBA copybook. Registering the ORBIX-STATUS-INFORMATION block allows the COBOL runtime to populate it with exception information, if necessary.  
You can use the ORBIX-STATUS-INFORMATION data item (in the CORBA copybook) to check the status of any Orbix call. The EXCEPTION-NUMBER numeric data item is important in this case. If this item is 0, it means the call was successful. Otherwise, EXCEPTION-NUMBER holds the system exception number that occurred. You should test this data item after any Orbix call.
2. ORBARGS is called to initialize a connection to the ORB.
3. ORBREG is called to register the IDL interface with the Orbix COBOL runtime.
4. The client reads the stringified object reference for the object from the PDS member that has been populated by the server. For the purposes of this example, the IOR member is contained in `orbixhlq.DEMOS.IORS(SIMPLE)`.
5. STRSET is called to create an unbounded string to which the stringified object reference is copied.
6. STRTOOBJ is called to create an object reference to the server object that is represented by the IOR. This must be done to allow operation invocations on the server. The STRTOOBJ 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.
7. After the object reference is created, ORBEXEC 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. The operation name must have at least one trailing space. The generated operation condition names found in the SIMPLE copybook already handle this.

The same argument description is used by the server, and is found in the `SIMPLE` copybook. For example, see  
`orbixhlq.DEMOS.COBOL.COPYLIB(SIMPLE).`

---

### **Location of the SIMPLECL program**

You can find a complete version of the `SIMPLECL` client program in  
`orbixhlq.DEMOS.COBOL.SRC(SIMPLECL).`

---

# Building and Running the Client

---

JCL to build the client	Sample JCL used to compile and link the client can be found in the third step of <code>orbixhlq.DEMOS.COBOL.BUILD.JCL(SIMPLECB)</code> .
Resulting load module	When the JCL has successfully executed, it results in a load module that is contained in <code>orbixhlq.DEMOS.COBOL.LOAD(SIMPLECL)</code> .
JCL to run the client	<p>Provided the IMS adapter has been started, you can use the following JCL to run the supplied <code>SIMPLECL</code> client application:</p> <pre>orbixhlq.DEMOS.COBOL.RUN.JCL(SIMPLECL)</pre> <p><b>Note:</b> See the <i>IMS Adapter Administrator's Guide</i> for details of how to start the IMS adapter.</p>
Client output	<p>The following is an example of the output produced by the batch client for the supplied demonstration:</p> <pre>Initializing the ORB Registering the Interface Reading object reference from file invoking Simple::call_me Simple demo complete.</pre>

# Getting Started in CICS

*This chapter introduces CICS application programming with Orbix E2A, by showing how to use Orbix E2A to develop a simple distributed application that features a COBOL client running in batch and a COBOL server running in CICS.*

**In this chapter**

This chapter discusses the following topics:

<a href="#">Overview and Setup Requirements</a>	<a href="#">page 84</a>
<a href="#">Developing the Application Interfaces</a>	<a href="#">page 89</a>
<a href="#">Developing the Server</a>	<a href="#">page 95</a>
<a href="#">Developing and Running the Client</a>	<a href="#">page 109</a>

**Note:** The example provided in this chapter requires use of the CICS adapter, which is supplied as part of the Orbix E2A Application Server Platform Mainframe Edition. See the *CICS Adapter Administrator's Guide* for more details about the CICS adapter.

---

# Overview and Setup Requirements

---

## Introduction

This section provides an overview of the main steps involved in creating an Orbix E2A COBOL 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 E2A COBOL application are:

Step	Action
1	<a href="#">“Developing the Application Interfaces” on page 89.</a>
2	<a href="#">“Developing the Server” on page 95.</a>
3	<a href="#">“Developing and Running the Client” on page 109.</a>

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

## The Simple demonstration

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

- An Orbix E2A COBOL server that implements a simple persistent POA-based server.
- An Orbix E2A COBOL 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 E2A Application Server Platform Mainframe Edition, because the client and server are written in the same language and running on the same platform.

---

**The demonstration server**

The server runs in a CICS region. It accepts and processes requests from the client across the network, and communicates with the client via the CICS adapter.

See [“Location of supplied code and JCL” on page 85](#) for details of where you can find an example of the supplied server. See [“Developing the Server” on page 95](#) for more details of how to develop the server.

---

**The demonstration client**

The client runs as a batch job. It 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 via the CICS adapter. When the operation has completed, a reply message is sent back to the client via the CICS adapter. This completes a single remote CORBA invocation.

See [“Location of supplied code and JCL” on page 85](#) for details of where you can find an example of the supplied client. See [“Developing and Running the Client” on page 109](#) 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 CICS `SIMPLE` demonstration have been provided with your installation. Apart from site-specific changes to some JCL, these do not require editing.

[Table 10](#) provides a summary of the code elements and JCL components that are relevant to the CICS `SIMPLE` demonstration (where *orbixhlq* represents your installation’s high-level qualifier).

**Table 10:** *Supplied Code and JCL (Sheet 1 of 2)*

Location	Description
<code>orbixhlq.DEMOS.IDL(SIMPLE)</code>	This is the supplied IDL.
<code>orbixhlq.DEMOS.CICS.COBOBOL.SRC(SIMPLESV)</code>	This is the source code for the CICS server mainline program, which is generated when you run the JCL in <code>orbixhlq.DEMOS.CICS.COBOBOL.BUILD.JCL(SIMPLIDL)</code> . (The CICS server mainline code is not shipped with the product. You must run the <code>SIMPLIDL</code> JCL to generate it.)
<code>orbixhlq.DEMOS.CICS.COBOBOL.SRC(SIMPLES)</code>	This is the source code for the CICS server implementation program.

**Table 10:** *Supplied Code and JCL (Sheet 2 of 2)*

Location	Description
<code>orbixhlq.DEMOS.COBOL.SRC(SIMPLECL)</code>	This is the source code for the batch client program.
<code>orbixhlq.DEMOS.CICS.COBOL.BUILD.JCL(SIMPLIDL)</code>	<p>This JCL runs the Orbix E2A IDL compiler, to generate COBOL source and copybooks for the CICS server. This JCL specifies the <code>-s</code> and <code>-TCICS</code> compiler arguments, to generate CICS server mainline code. It does not specify the <code>-z</code> argument, which generates server implementation code.</p> <p>This JCL also specifies the <code>-mfa</code> and <code>-ttran_or_program_name</code> arguments, to generate the adapter mapping file, which is then written to <code>orbixhlq.DEMOS.CICS.MFAMAP(SIMPLEA)</code>. The contents of the <code>SIMPLEA</code> member are <code>(Simple/SimpleObject,call_me,SIMPLESV)</code> (that is, fully qualified interface name followed by operation name followed by CICS APPC transaction name or CICS EXCI program name). See the <i>CICS Adapter Administrator's Guide</i> for more details about generating adapter mapping files.</p>
<code>orbixhlq.DEMOS.COBOL.BUILD.JCL(SIMPLECB)</code>	This JCL compiles the client program.
<code>orbixhlq.DEMOS.CICS.COBOL.BUILD.JCL(SIMPLESB)</code>	This JCL compiles and links the CICS server mainline and CICS server implementation programs.
<code>orbixhlq.DEMOS.CICS.COBOL.BUILD.JCL(SIMPLREG)</code>	This JCL registers the IDL in the Interface Repository.
<code>orbixhlq.DEMOS.CICS.COBOL.BUILD.JCL(SIMPLIOR)</code>	This JCL obtains the IOR that the client of the CICS server requires to locate the CICS adapter.
<code>orbixhlq.DEMOS.COBOL.RUN.JCL(SIMPLECL)</code>	This JCL runs the batch client.

**Note:** Other code elements and JCL components are provided for the batch and IMS versions of the `SIMPLE` demonstration. See [“Getting Started in Batch” on page 13](#) and [“Getting Started in IMS” on page 49](#) for more details of these.



## Supplied copybooks

Table 11 provides a summary of the various copybooks supplied with your product installation that are relevant to CICS. In Table 11, *servers* means CICS servers, and *clients* means batch clients. Again, *orbixhlq* represents your installation's high-level qualifier.

**Table 11:** *Supplied Copybooks (Sheet 1 of 2)*

Location	Description
<code>orbixhlq.INCLUDE.COPYLIB(CHKERRS)</code>	This is used by clients. It is not used by CICS servers. It contains a COBOL function that can be called by the batch client, to check if an exception has occurred and report it.
<code>orbixhlq.INCLUDE.COPYLIB(CERRSMFA)</code>	This is used by servers. It contains a COBOL function that can be called by the CICS server, to check if an exception has occurred and report it.
<code>orbixhlq.INCLUDE.COPYLIB(CORBA)</code>	This is used both by clients and servers. It contains various Orbix E2A COBOL definitions, such as <code>REQUEST-INFO</code> used by the <code>COAREQ</code> function, and <code>ORBIX-STATUS-INFORMATION</code> which is used to register and report exceptions raised by the COBOL runtime.
<code>orbixhlq.INCLUDE.COPYLIB(CORBATYP)</code>	This is used both by clients and servers. It contains the COBOL typecode representation for IDL basic types.
<code>orbixhlq.INCLUDE.COPYLIB(IORSLCT)</code>	This is used by clients. It is not used by CICS servers. It contains the COBOL <code>SELECT</code> statement entry for file processing, for use with the <code>COPY...REPLACING</code> statement.
<code>orbixhlq.INCLUDE.COPYLIB(IORFD)</code>	This is used by clients. It is not used by CICS servers. It contains the COBOL <code>FD</code> statement entry for file processing, for use with the <code>COPY...REPLACING</code> statement.
<code>orbixhlq.INCLUDE.COPYLIB(CHKFILE)</code>	This is used by clients. It is not used by CICS servers. It is used for file handling error checking.

**Table 11:** *Supplied Copybooks (Sheet 2 of 2)*

Location	Description
<code>orbixhlq.INCLUDE.COPYLIB(PROCPARM)</code>	This is used by clients. It is not used by CICS servers. It contains the appropriate definitions for a COBOL program to accept parameters from the JCL for use with the ORBARGS API (that is, the <code>argument-string</code> parameter).
<code>orbixhlq.DEMOS.CICS.COBOLE.COPYLIB</code>	This PDS is used to store all CICS copybooks that are generated by the Orbix E2A IDL compiler when you run the supplied <code>SIMPLIDL</code> JCL for the CICS demonstration. It also contains copybooks with Working Storage data definitions and Procedure Division paragraphs for use with the nested sequences demonstration.
<code>orbixhlq.DEMOS.CICS.MFAMAP</code>	This PDS is empty at installation time. It is used to store the CICS adapter mapping member generated by the Orbix E2A IDL compiler when you run the supplied <code>SIMPLIDL</code> JCL.

**Note:** Some other copybooks are provided specifically for use with IMS. See [“Getting Started in IMS” on page 49](#) for more details of these.

### Checking JCL components

When creating the `SIMPLE` application, check that each step involved within the separate JCL components completes with a condition code of zero. If the condition codes are not zero, 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 COBOL source and copybooks from IDL interfaces, and provides a description of the members generated from the supplied `SimpleObject` interface.

---

## Steps to develop application interfaces

The steps to develop the interfaces to your application are:

Step	Action
1	Define public IDL interfaces to the objects required in your system. See <a href="#">“Defining IDL Interfaces” on page 90</a> .
2	Use the Orbix E2A IDL compiler to generate COBOL source code and copybooks from the defined IDL. See <a href="#">“Generating COBOL Source and Copybooks” on page 91</a> .

---

## Defining IDL Interfaces

### Defining the IDL

The first step in writing an Orbix E2A program is to define the IDL interfaces for the objects required in your system. The following is an example of the IDL for the `SimpleObject` interface that is supplied in

`orbixhlq.DEMOS.IDL(SIMPLE)`:

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

### Explanation of the IDL

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

### How the demonstration uses this IDL

For the purposes of this example, the `SimpleObject` CORBA object is implemented in COBOL in the supplied `SIMPLES` 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 interoperable object reference (IOR) from the relevant PDS member. The client invokes the `call_me()` operation on the `SimpleObject` object, and then exits.

---

## Generating COBOL Source and Copybooks

---

### The Orbix E2A IDL compiler

You can use the Orbix E2A IDL compiler to generate COBOL source and copybooks from IDL definitions.

---

### Orbix E2A IDL compiler configuration

The Orbix E2A IDL compiler uses the Orbix E2A configuration member for its settings. The `SIMPLIDL` JCL that runs the compiler uses a configuration member provided in `orbixhlq.CONFIG(IDL)`. See [“Orbix E2A IDL Compiler” on page 237](#) for more details.

---

### Running the Orbix E2A IDL compiler

The COBOL source for the CICS server demonstration described in this chapter is generated in the first step of the following job:

```
orbixhlq.DEMOS.CICS.COBOL.BUILD.JCL(SIMPLIDL)
```

Generated source code members

Table 12 shows the server source code members that the Orbix E2A IDL compiler generates, based on the defined IDL.

**Table 12:** *Generated Server Source Code Members*

Member	JCL Keyword Parameter	Description
<i>idlmembernameS</i>	IMPL	This is the CICS server implementation source code member. It contains stub paragraphs for all the callable operations.  This is only generated if you specify both the <code>-z</code> and <code>-TCICS</code> arguments with the IDL compiler.
<i>idlmembernameSV</i>	IMPL	This is the CICS server mainline source code member.  This is only generated if you specify both the <code>-s</code> and <code>-TCICS</code> arguments with the IDL compiler.

**Note:** For the purposes of this example, the `SIMPLES` server implementation is already provided in your product installation. Therefore, the `-z` IDL compiler argument used to generate it is not specified in the supplied `SIMPLIDL` JCL. The `SIMPLESV` server mainline is not already provided, so the `-s` argument used to generate it is specified in the supplied JCL. See [“Orbix E2A IDL Compiler” on page 237](#) for more details of the `-s`, `-z`, and `-TCICS` arguments used to generate CICS server code.

## Generated COBOL copybooks

[Table 13](#) shows the COBOL copybooks that the Orbix E2A IDL compiler generates, based on the defined IDL.

**Table 13:** *Generated COBOL Copybooks*

Copybook	JCL Keyword Parameter	Description
<i>idlmembername</i>	COPYLIB	This copybook contains data definitions that are used for working with operation parameters and return values for each interface defined in the IDL member.  The name for this copybook does not take a suffix.
<i>idlmembernameX</i>	COPYLIB	This copybook contains data definitions that are used by the COBOL runtime to support the interfaces defined in the IDL member.  This copybook is automatically included in the <i>idlmembername</i> copybook.
<i>idlmembernameD</i>	COPYLIB	This copybook contains procedural code for performing the correct paragraph for the requested operation.  This copybook is automatically included in the <i>idlmembernameS</i> source code member.

## How IDL maps to COBOL copybooks

Each IDL interface maps to a group of COBOL data definitions. There is one definition for each IDL operation. A definition contains each of the parameters for the relevant IDL operation in their corresponding COBOL representation. See [“IDL-to-COBOL Mapping” on page 161](#) for details of how IDL types map to COBOL.

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

Generated adapter mapping member

.



---

# Developing the Server

---

## Overview

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

---

## Steps to develop the server

The steps to develop the server application are:

Step	Action
1	<a href="#">“Writing the Server Implementation” on page 96.</a>
2	<a href="#">“Writing the Server Mainline” on page 100.</a>
3	<a href="#">“Building the Server” on page 104.</a>
4	<a href="#">“Preparing the Server to Run in CICS” on page 105.</a>

# Writing the Server Implementation

## The server implementation program

You must implement the server interface by writing a COBOL program that implements each operation in the *idlmembername* copybook. For the purposes of this example, you must write a COBOL program that implements each operation in the *SIMPLE* copybook. When you specify the *-z* and *-TCICS* arguments with the Orbix E2A IDL compiler in this case, it generates a skeleton program called *SIMPLES*, which is a useful starting point.

## Example of the CICS SIMPLES program

The following is an example of the CICS *SIMPLES* program:

### Example 7: *The CICS SIMPLES Demonstration (Sheet 1 of 2)*

```
*****
* Identification Division
*****
IDENTIFICATION DIVISION.
PROGRAM-ID.          SIMPLES.

ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY SIMPLE.
COPY CORBA.

01 WS-INTERFACE-NAME          PICTURE X(30).
01 WS-INTERFACE-NAME-LENGTH   PICTURE 9(09) BINARY
                               VALUE 30.

*****
* Procedure Division
*****
PROCEDURE DIVISION.

1      ENTRY "DISPATCH".

2      CALL "COAREQ"          USING REQUEST-INFO.
      SET WS-COAREQ TO TRUE.
      PERFORM CHECK-STATUS.
```

### Example 7: The CICS SIMLES Demonstration (Sheet 2 of 2)

```
3  * Resolve the pointer reference to the interface name which is
   * the fully scoped interface name
   * Note make sure it can handle the max interface name length
      CALL "STRGET"      USING INTERFACE-NAME
                              WS-INTERFACE-NAME-LENGTH
                              WS-INTERFACE-NAME.

      SET WS-STRGET TO TRUE.
      PERFORM CHECK-STATUS.

*****
* Interface(s)  evaluation:
*****
      MOVE SPACES TO SIMPLE-SIMPLEOBJECT-OPERATION.

      EVALUATE WS-INTERFACE-NAME
      WHEN 'IDL:Simple/SimpleObject:1.0'

4  * Resolve the pointer reference to the operation information
      CALL "STRGET" USING OPERATION-NAME
                              SIMPLE-S-3497-OPERATION-LENGTH
                              SIMPLE-SIMPLEOBJECT-OPERATION

      SET WS-STRGET TO TRUE
      PERFORM CHECK-STATUS
      DISPLAY  "Simple::" SIMPLE-SIMPLEOBJECT-OPERATION
              "invoked"
      END-EVALUATE.

5  COPY SIMPLED.

      GOBACK.

6  DO-SIMPLE-SIMPLEOBJECT-CALL-ME.
      CALL "COAGET"      USING SIMPLE-SIMPLEOBJECT-70FE-ARGS.
      SET WS-COAGET TO TRUE.
      PERFORM CHECK-STATUS.

      CALL "COAPUT"      USING SIMPLE-SIMPLEOBJECT-70FE-ARGS.
      SET WS-COAPUT TO TRUE.
      PERFORM CHECK-STATUS.

*****
* Check Errors Copybook
*****

7  COPY CERRSMFA.
```

## Explanation of the CICS SIMPLES program

---

The SIMPLES program can be explained as follows:

1. The DISPATCH logic is automatically coded for you, and the bulk of the code is contained in the SIMPLED copybook. When an incoming request arrives from the network, it is processed by the ORB and a call is made to the DISPATCH entry point.
2. COAREQ is called to provide information about the current invocation request, which is held in the REQUEST-INFO block that is contained in the CORBA copybook.  
COAREQ is called once for each operation invocation—after a request has been dispatched to the server, but before any calls are made to access the parameter values.
3. STRGET is called to copy the characters in the unbounded string pointer for the interface name to the string item representing the fully scoped interface name.
4. STRGET is called again to copy the characters in the unbounded string pointer for the operation name to the string item representing the operation name.
5. The procedural code used to perform the correct paragraph for the requested operation is copied into the program from the SIMPLED copybook.
6. Each operation has skeleton code, with appropriate calls to COAPUT and COAGET to copy values to and from the COBOL structures for that operation's argument list. You must provide a correct implementation for each operation. You must call COAGET and COAPUT, even if your operation takes no parameters and returns no data. You can simply pass in a dummy area as the parameter list.
7. The CICS server implementation uses a COPY CERRSMFA statement instead of COPY CHKERRS.

**Note:** The supplied SIMPLES program is only a suggested way of implementing an interface. It is not necessary to have all operations implemented in the same COBOL program.

**Location of the CICS SIMPLES  
program**

---

You can find a complete version of the CICS `SIMPLES` server implementation program in

# Writing the Server Mainline

## The server mainline program

The next step is to write the server mainline program in which to run the server implementation. For the purposes of this example, when you specify the `-s` and `-TCICS` arguments with the Orbix E2A IDL compiler, it generates a program called `SIMPLESV`, 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 adapter.

## Example of the CICS SIMPLESV program

The following is an example of the CICS `SIMPLESV` program:

### Example 8: *The CICS SIMPLESV Demonstration (Sheet 1 of 3)*

```
IDENTIFICATION DIVISION.
PROGRAM-ID.          SIMPLESV.
ENVIRONMENT DIVISION.
DATA DIVISION.

WORKING-STORAGE SECTION.

COPY SIMPLE.
COPY CORBA.

01 ARG-LIST                                PICTURE X(01)
                                           VALUE SPACES.
01 ARG-LIST-LEN                            PICTURE 9(09) BINARY
                                           VALUE 0.
01 ORB-NAME                               PICTURE X(10)
                                           VALUE "simple_orb".
01 ORB-NAME-LEN                           PICTURE 9(09) BINARY
                                           VALUE 10.

01 SERVER-NAME                             PICTURE X(07)
                                           VALUE "simple ".
01 SERVER-NAME-LEN                        PICTURE 9(09) BINARY
                                           VALUE 6.

01 INTERFACE-LIST.
```

**Example 8:** *The CICS SIMPLESV Demonstration (Sheet 2 of 3)*

```
      03 FILLER                                PICTURE X(28)
          VALUE "IDL:Simple/SimpleObject:1.0 ".
01 INTERFACE-NAMES-ARRAY REDEFINES INTERFACE-LIST.
      03 INTERFACE-NAME OCCURS 1 TIMES          PICTURE X(28).

01 OBJECT-ID-LIST.
      03 FILLER                                PICTURE X(27)
          VALUE "Simple/SimpleObject_object ".
01 OBJECT-ID-ARRAY REDEFINES OBJECT-ID-LIST.
      03 OBJECT-IDENTIFIER OCCURS 1 TIMES      PICTURE X(27).

*****
* Object values for the Interface(s)
*****
01 SIMPLE-SIMPLEOBJECT-OBJ                      POINTER
          VALUE NULL.

PROCEDURE DIVISION.

INIT.

1      CALL "ORBSTAT"    USING ORBIX-STATUS-INFORMATION.
      SET WS-ORBSTAT TO TRUE.
      PERFORM CHECK-STATUS.

2      CALL "ORBARGS"    USING ARG-LIST
          ARG-LIST-LEN
          ORB-NAME
          ORB-NAME-LEN.

      SET WS-ORBARGS TO TRUE.
      PERFORM CHECK-STATUS.

3      CALL "ORBSVR"     USING SERVER-NAME
          SERVER-NAME-LEN.

      SET WS-ORBSVR TO TRUE.
      PERFORM CHECK-STATUS.

*****
* Interface Section Block
*****

* Generating Object Reference for interface Simple/SimpleObject

4      CALL "ORBREG"     USING SIMPLE-SIMPLEOBJECT-INTERFACE.
```

**Example 8:** *The CICS SIMPLESV Demonstration (Sheet 3 of 3)*

```

SET WS-ORBREG TO TRUE.
PERFORM CHECK-STATUS.

5 CALL "OBJNEW" USING SERVER-NAME
                                INTERFACE-NAME OF INTERFACE-NAMES-ARRAY(1)
                                OBJECT-IDENTIFIER OF OBJECT-ID-ARRAY(1)
                                SIMPLE-SIMPLEOBJECT-OBJ.

SET WS-OBJNEW TO TRUE.
PERFORM CHECK-STATUS.

6 CALL "COARUN".
SET WS-COARUN TO TRUE.
PERFORM CHECK-STATUS.

EXIT-PRG.
GOBACK.

*****
* Check Errors Copybook
*****
COPY CERRSMFA.

```

**Explanation of the CICS  
SIMPLESV program**

The SIMPLESV program can be explained as follows:

1. ORBSTAT is called to register the ORBIX-STATUS-INFORMATION block that is contained in the CORBA copybook. Registering the ORBIX-STATUS-INFORMATION block allows the COBOL runtime to populate it with exception information, if necessary.
2. ORBARGS is called to initialize a connection to the ORB.
3. ORBSRV is called to set the server name.
4. ORBREG is called to register the IDL interface, SimpleObject, with the Orbix COBOL runtime.
5. OBJNEW is called to create a persistent server object of the SimpleObject type, with an object ID of my\_simple\_object.
6. COARUN 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.



---

**Location of the CICS SIMPLESV  
program**

You can find a complete version of the CICS SIMPLESV server mainline program in *orbixhlq.DEMOS.CICS.COBOL.SRC(SIMPLESV)* after you have run the supplied SIMPLIDL JCL to run the Orbix E2A IDL compiler.

---

## Building the Server

---

### Location of the JCL

Sample JCL used to compile and link the CICS server mainline and server implementation is in `orbixhlq.DEMOS.CICS.COBOLOAD.BUILD.JCL(SIMPLESB)`.

---

### Resulting load module

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

---

# Preparing the Server to Run in CICS

Overview

This section describes the required steps to allow the server to run in a CICS MPP region. When all the steps in this section have been completed, the server is started automatically within CICS, as required.

Steps

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

Step	Action
1	Define an APPC transaction definition or EXCI program definition for CICS.
2	Provide the CICS server load module to a CICS region.
3	Generate mapping file entries for the CICS adapter.
4	Add the IDL to the Interface Repository.
5	Obtain the IOR for use by the client program.

Step 1—Defining program or transaction definition for CICS

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

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

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

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

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

---

### Step 2—Providing load module to CICS region

Ensure that the *orbixhlq.DEMOS.CICS.COBOLOAD* 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 file entries

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

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

Adapter mapping file generation is performed as part of the supplied *orbixhlq.DEMOS.CICS.COBOBUILD.JCL(SIMPLIDL) JCL*. The *-mfa: -ttran\_or\_program\_name* argument with the IDL compiler generates the mapping file. 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 file is generated. See the *CICS Adapter Administrator's Guide* for full details about adapter mapping files.

#### Step 4—Adding IDL to Interface Repository

The CICS adapter needs to be able to obtain the IDL for the COBOL server from the Interface Repository, so that it knows what data types it has to marshal into CICS for the server, and what data types it can expect back from the CICS APPC transaction or CICS EXCI program. Ensure that the relevant IDL for the server has been added to the Interface Repository before the CICS adapter is started.

To add IDL to the Interface Repository, the Interface Repository must be running. You can use the JCL in *orbixhlq.JCL(IFR)* to start it. The Interface Repository uses the Orbix E2A configuration member for its settings. The JCL that you use to start the Interface Repository uses a configuration member that is provided in *orbixhlq.DOMAINS(FILEDOMA)*.

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

```
//          JCLLIB ORDER=(HLQ.ASP50.PROCS)
//          INCLUDE MEMBER=(ORXVARS)
//IDLCBL    EXEC ORXIDL,
//          SOURCE=SIMPLE,
//          IDL=&ORBIX..DEMOS.IDL,
//          COPYLIB=&ORBIX..DEMOS.CICS.COBOLE.COPYLIB,
//          IMPL=&ORBIX..DEMOS.CICS.COBOLE.SRC,
//          IDLPARM=' -R '
```

## Step 5—Obtaining the adapter IOR

The final step is to use the `resolve` command, to obtain the IOR that the client needs to locate the CICS adapter. To obtain the CICS adapter's IOR, the following prerequisites apply:

- The Interface Repository must be running and contain the relevant IDL. See [“Step 4—Adding IDL to Interface Repository” on page 107](#) for details of how to start it, if it is not already running.
- The CICS adapter must be running. See the *CICS Adapter Administrator's Guide* for more details about it and how to start it.
- The CICS adapter mapping file must contain the relevant mapping entries. For the purposes of this example, ensure that the `orbixhlq.DEMOS.CICS.MFAMAP(SIMPLEA)` mapping member is being used. See the *CICS Adapter Administrator's Guide* for details about adapter mapping files.

The following JCL, which includes the `resolve` command to obtain the IOR, is supplied in `orbixhlq.DEMOS.CICS.COBOLE.BUILD.JCL(SIMPLIOR)`.

```
//          JCLLIB ORDER=(HLQ.ASP50.PROCS)
//          INCLUDE MEMBER=(ORXVARS)
//REG      EXEC PROC=ORXADMIN,
//  PPARM='mfa resolve Simple/SimpleObject > DD:IOR'
//IOR DD DSN=&ORBIX..DEMOS.IORS(SIMPLE),DISP=SHR
```

---

# Developing and Running the Client

---

## Overview

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

**Note:** The Orbix E2A IDL compiler does not generate COBOL client stub code.

## Steps to develop the client

The steps to develop and run the client application are:

Step	Action
1	<a href="#">"Writing the Client" on page 110.</a>
2	<a href="#">"Building and Running the Client" on page 115.</a>

---

# Writing the Client

## The client program

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

## Example of the `SIMPLECL` program

The following is an example of the `SIMPLECL` program:

### Example 9: *The SIMPLECL Demonstration Program (Sheet 1 of 3)*

```
IDENTIFICATION DIVISION.  
PROGRAM-ID.                                SIMPLECL.  
  
ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.  
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
    COPY IORSLCT REPLACING  
        "X-IOR" BY SIMPLE-SIMPLEOBJECT-IOR  
        "X-IORFILE" BY "IORFILE"  
        "X-IOR-STAT" BY SIMPLE-SIMPLEOBJECT-IOR-STAT.  
DATA DIVISION.  
FILE SECTION.  
  
    COPY IORFD REPLACING  
        "X-IOR" BY SIMPLE-SIMPLEOBJECT-IOR  
        "X-REC" BY SIMPLE-SIMPLEOBJECT-REC.  
  
WORKING-STORAGE SECTION.  
  
COPY SIMPLE.  
COPY CORBA.  
  
01 WS-SIMPLE-IOR                                PICTURE X(2048).  
01 SIMPLE-IOR-LENGTH                            PICTURE 9(9) BINARY  
                                                VALUE 2048.  
01 SIMPLE-SIMPLEOBJECT-IOR-STAT                PICTURE 9(02).  
01 SIMPLE-SIMPLEOBJECT-OBJ                    POINTER  
                                                VALUE NULL.  
01 ARG-LIST                                    PICTURE X(1)  
                                                VALUE SPACES.  
01 ARG-LIST-LEN                                PICTURE 9(09) BINARY  
                                                VALUE 0.
```



**Example 9:** *The SIMPLECL Demonstration Program (Sheet 2 of 3)*

```
01 ORB-NAME                PICTURE X(10)
                           VALUE "simple_orb".
01 ORB-NAME-LEN            PICTURE 9(09) BINARY
                           VALUE 10.
01 IOR-REC-PTR             POINTER
                           VALUE NULL.
01 IOR-REC-LEN            PICTURE 9(09) BINARY
                           VALUE 2048.

COPY PROCPARM.
1   CALL "ORBSTAT" USING ORBIX-STATUS-INFORMATION.

*   ORB initialization
    DISPLAY "Initializing the ORB".
2   CALL "ORBARGS"  USING ARG-LIST
                           ARG-LIST-LEN
                           ORB-NAME
                           ORB-NAME-LEN.

    SET WS-ORBARGS TO TRUE.
    PERFORM CHECK-STATUS.

*   Register interface TypeTest
    DISPLAY "Registering the Interface".
3   CALL "ORBREG"  USING SIMPLE-SIMPLEOBJECT-INTERFACE.
    SET WS-ORBREG TO TRUE.
    PERFORM CHECK-STATUS.

*
4  **   Read in the IOR from a file which has been populated
  **   by the server program.
  *

    OPEN INPUT SIMPLE-SIMPLEOBJECT-IOR.
    COPY CHKFILE REPLACING
      "X-IOR-STAT" BY SIMPLE-SIMPLEOBJECT-IOR-STAT.

    DISPLAY "Reading object reference from file".
    READ SIMPLE-SIMPLEOBJECT-IOR.
    COPY CHKFILE REPLACING
      "X-IOR-STAT" BY SIMPLE-SIMPLEOBJECT-IOR-STAT.

    MOVE SIMPLE-SIMPLEOBJECT-REC TO WS-SIMPLE-IOR.

*   IOR Record read successfully
    CLOSE SIMPLE-SIMPLEOBJECT-IOR.
    COPY CHKFILE REPLACING
```

**Example 9:** *The SIMPLECL Demonstration Program (Sheet 3 of 3)*

```

        "X-IOR-STAT" BY SIMPLE-SIMPLEOBJECT-IOR-STAT.
5  * Set the COBOL pointer to point to the IOR string
    CALL "STRSET"      USING IOR-REC-PTR
                                IOR-REC-LEN
                                WS-SIMPLE-IOR.

    SET WS-STRSET TO TRUE.
    PERFORM CHECK-STATUS.

6  * Obtain object reference from the IOR
    CALL "STRTOOBJ"    USING IOR-REC-PTR
                                SIMPLE-SIMPLEOBJECT-OBJ

    SET WS-STRTTOOBJ TO TRUE.
    PERFORM CHECK-STATUS.

    * Releasing the memory
    CALL "STRFREE"     USING IOR-REC-PTR.
    SET WS-STRFREE TO TRUE.
    PERFORM CHECK-STATUS.

    SET SIMPLE-SIMPLEOBJECT-CALL-ME TO TRUE
    DISPLAY "invoking Simple::" SIMPLE-SIMPLEOBJECT-OPERATION.

7  CALL "ORBEXEC"      USING SIMPLE-SIMPLEOBJECT-OBJ
                                SIMPLE-SIMPLEOBJECT-OPERATION
                                SIMPLE-SIMPLEOBJECT-70FE-ARGS
                                SIMPLE-USER-EXCEPTIONS.

    SET WS-ORBEXEC TO TRUE.
    PERFORM CHECK-STATUS.

    CALL "OBJREL"      USING SIMPLE-SIMPLEOBJECT-OBJ.
    SET WS-OBJREL TO TRUE.
    PERFORM CHECK-STATUS.

    DISPLAY "Simple demo complete.".

EXIT-PRG.
*=====
STOP RUN.

*****
* Check Errors Copybook
*****
        COPY CHKERRS.

```

## Explanation of the SIMPLECL program

---

The SIMPLECL program can be explained as follows:

1. ORBSTAT is called to register the ORBIX-STATUS-INFORMATION block that is contained in the CORBA copybook. Registering the ORBIX-STATUS-INFORMATION block allows the COBOL runtime to populate it with exception information, if necessary.  
You can use the ORBIX-STATUS-INFORMATION data item (in the CORBA copybook) to check the status of any Orbix call. The EXCEPTION-NUMBER numeric data item is important in this case. If this item is 0, it means the call was successful. Otherwise, EXCEPTION-NUMBER holds the system exception number that occurred. You should test this data item after any Orbix call.
2. ORBARGS is called to initialize a connection to the ORB.
3. ORBREG is called to register the IDL interface with the Orbix COBOL runtime.
4. The client reads the stringified object reference for the object from the PDS member that has been populated by the server. For the purposes of this example, the IOR member is contained in `orbixhlq.DEMOS.IORS(SIMPLE)`.
5. STRSET is called to create an unbounded string to which the stringified object reference is copied.
6. STRTOOBJ is called to create an object reference to the server object that is represented by the IOR. This must be done to allow operation invocations on the server. The STRTOOBJ 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.
7. After the object reference is created, ORBEXEC 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. The operation name must have at least one trailing space. The generated operation condition names found in the SIMPLE copybook already handle this.

The same argument description is used by the server, and is found in the `SIMPLE` copybook. For example, see  
`orbixhlq.DEMOS.COBOL.COPYLIB(SIMPLE).`

---

### **Location of the SIMPLECL program**

You can find a complete version of the `SIMPLECL` client program in  
`orbixhlq.DEMOS.COBOL.SRC(SIMPLECL).`

---

## Building and Running 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.COBOL.BUILD.JCL(SIMPLECB)`.

---

### Resulting load module

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

---

### JCL to run the client

Provided the CICS adapter has been started, you can use the following JCL to run the supplied `SIMPLECL` client application:

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

**Note:** See the *CICS Adapter Administrator's Guide* for details of how to start the CICS adapter.

---

### Client output

The following is an example of the output produced by the batch client for the supplied demonstration:

```
Initializing the ORB
Registering the Interface
Reading object reference from file
invoking Simple::call_me
Simple demo complete.
```



# 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:

<a href="#">IDL</a>	<a href="#">page 118</a>
<a href="#">Modules and Name Scoping</a>	<a href="#">page 119</a>
<a href="#">Interfaces</a>	<a href="#">page 120</a>
<a href="#">IDL Data Types</a>	<a href="#">page 140</a>
<a href="#">Defining Data Types</a>	<a href="#">page 154</a>

---

# IDL

## Overview

An IDL-defined object can be implemented in any language that IDL maps to, including C++, Java, COBOL, and PL/I. 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, COBOL, and PL/I. Each IDL mapping specifies how an IDL interface corresponds to a language-specific implementation. The Orbix E2A IDL compiler uses these mappings to convert IDL definitions to language-specific definitions that conform to the semantics of that language.

## Overall structure

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

## IDL definition structure

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

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

  interface Account {
    //...
  };
};
```



---

# Modules and Name Scoping

---

## Resolving a name

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

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

---

## Referencing interfaces

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

*module-name::interface-name*

For example, the fully scoped names of the `Bank` and `Account` interfaces shown in [“IDL definition structure” on page 118](#) are, respectively, `BankDemo::Bank` and `BankDemo::Account`.

---

## Nesting restrictions

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

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

---

# Interfaces

**In this section**

The following topics are discussed in this section:

<a href="#">Interface Contents</a>	<a href="#">page 122</a>
<a href="#">Operations</a>	<a href="#">page 123</a>
<a href="#">Attributes</a>	<a href="#">page 126</a>
<a href="#">Exceptions</a>	<a href="#">page 127</a>
<a href="#">Empty Interfaces</a>	<a href="#">page 128</a>
<a href="#">Inheritance of Interfaces</a>	<a href="#">page 129</a>
<a href="#">Multiple Inheritance</a>	<a href="#">page 130</a>

**Overview**

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

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

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

**Operations and attributes**

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

- Operations of an interface give clients access to an object's behavior. When a client invokes an operation on an object, it sends a message to that object. The ORB transparently dispatches the call to the object,

whether it is in the same address space as the client, in another address space on the same machine, or in an address space on a remote machine.

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

---

### Account interface IDL sample

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

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

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

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

---

### Code explanation

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

---

## Interface Contents

---

### IDL interface components

An IDL interface definition typically has the following components.

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

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

---

# Operations

## Overview

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

## Operation components

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

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

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

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

## Operations IDL sample

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

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

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

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

Code explanation	On each invocation, both operations expect the client to supply an argument for the <code>amount</code> parameter, and return <code>void</code> . Invocations on the <code>withdraw()</code> operation can also raise the <code>InsufficientFunds</code> 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 COBOL runtime uses parameter-passing modes to determine in which direction or directions it must marshal a parameter.						
Parameter-passing mode qualifiers	<p>There are three parameter-passing mode qualifiers:</p> <table><tr><td><code>in</code></td><td>This means that the parameter is initialized only by the client and is passed to the object.</td></tr><tr><td><code>out</code></td><td>This means that the parameter is initialized only by the object and returned to the client.</td></tr><tr><td><code>inout</code></td><td>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.</td></tr></table> <p>In general, you should avoid using <code>inout</code> parameters. Because an <code>inout</code> 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, <code>in</code> and <code>out</code>, the caller can decide for itself when to discard the parameter.</p>	<code>in</code>	This means that the parameter is initialized only by the client and is passed to the object.	<code>out</code>	This means that the parameter is initialized only by the object and returned to the client.	<code>inout</code>	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.
<code>in</code>	This means that the parameter is initialized only by the client and is passed to the object.						
<code>out</code>	This means that the parameter is initialized only by the object and returned to the client.						
<code>inout</code>	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.						
One-way operations	By default, IDL operations calls are <i>synchronous</i> —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 <code>oneway</code> , a client that calls the operation remains unblocked while the object processes the call.						

**Note:** The COBOL runtime does not support one-way operations.

The COBOL runtime cannot guarantee the success of a one-way operation call. Because one-way operations do not support return data to the client, the client cannot ascertain the outcome of its invocation. The COBOL runtime indicates failure of a one-way operation only if the call fails before it exits the client's address space; in this case, the COBOL runtime raises a system exception.

A client can also issue non-blocking, or asynchronous, invocations. Refer to the *CORBA Programmer's Guide, C++* for more details.

---

### One-way operation constraints

Three constraints apply to a one-way operation:

- ◆ The return value must be set to `void`.
- ◆ Directions of all parameters must be set to `in`.
- ◆ No `raises` clause is allowed.

---

### One-way operation IDL sample

In the following example, the `Account` interface defines a one-way operation that sends a notice to an `Account` object:

```
module BankDemo {  
    //...  
    interface Account {  
        oneway void notice(in string text);  
        //...  
    };  
};
```

---

## Attributes

### Overview

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

### Qualified and unqualified attributes

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

### IDL readonly attributes sample

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

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

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

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

### Code explanation

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



# Exceptions

## IDL and exceptions

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

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

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

## The raises clause

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

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

## Example of IDL-defined exceptions

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

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

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

---

## Empty Interfaces

### Defining empty interfaces

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

### IDL empty interface sample

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

```
module PortableServer
{
    interface ServantManager {};

    interface ServantActivator : ServantManager {
        //...
    };

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

---

# Inheritance of Interfaces

## Inheritance overview

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

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

## Inheritance interface IDL sample

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

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

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

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

## Code sample explanation

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

---

## Multiple Inheritance

### Multiple inheritance IDL sample

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

```
module BankDemo {  
    interface Account {  
        //...  
    };  
  
    interface CheckingAccount : Account {  
        //...  
    };  
  
    interface SavingsAccount : Account {  
        //...  
    };  
  
    interface PremiumAccount :  
        CheckingAccount, SavingsAccount {  
        //...  
    };  
};
```

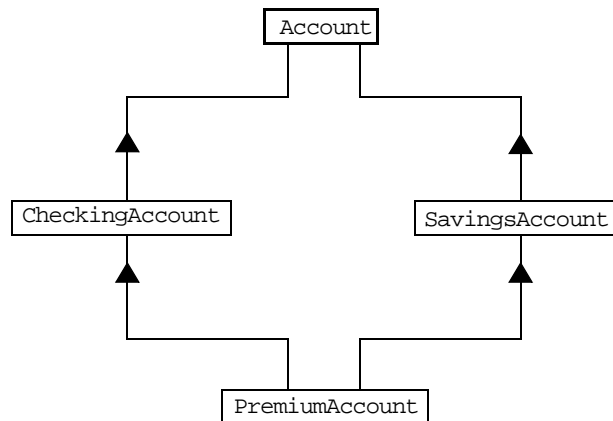
### Multiple inheritance constraints

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

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

**Inheritance hierarchy diagram**

Figure 4 shows the inheritance hierarchy for the `Account` interface, which is defined in [“Multiple inheritance IDL sample”](#) on page 130.



**Figure 4:** *Inheritance Hierarchy for PremiumAccount Interface*

---

## Inheritance of the Object Interface

---

### User-defined interfaces

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

---

### Object locator IDL sample

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

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

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

---

# Inheritance Redefinition

## Overview

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

## Inheritance redefinition IDL sample

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

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

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

## Forward Declaration of IDL Interfaces

### Overview

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

### Forward declaration IDL sample

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

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

    // Forward declaration of Account
    interface Account;

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

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

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

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



---

## Local Interfaces

### Overview

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

**Note:** The COBOL runtime and the Orbix E2A IDL compiler do not support local interfaces.

### Characteristics

Local interfaces differ from unconstrained interfaces in the following ways:

- A local interface can inherit from any interface, whether local or unconstrained. Unconstrained interfaces cannot inherit from local interfaces.
- Any non-interface type that uses a local interface is regarded as a local type. For example, a struct that contains a local interface member is regarded as a local struct, and is subject to the same localization constraints as a local interface.
- Local types can be declared as parameters, attributes, return types, or exceptions only in a local interface, or as state members of a valuetype.
- Local types cannot be marshalled, and references to local objects cannot be converted to strings through `ORB::object_to_string()`. Any attempts to do so throw a `CORBA::MARSHAL` exception.
- Any operation that expects a reference to a remote object cannot be invoked on a local object. For example, you cannot invoke any DII operations or asynchronous methods on a local object; similarly, you cannot invoke pseudo-object operations such as `is_a()` or `validate_connection()`. Any attempts to do so throw a `CORBA::NO_IMPLEMENT` exception.
- The ORB does not mediate any invocations on a local object. Thus, local interface implementations are responsible for providing the parameter copy semantics that a client expects.

- Instances of local objects that the OMG defines, as supplied by ORB products, are exposed either directly or indirectly through `ORB::resolve_initial_references()`.

**Implementation**

Local interfaces are implemented by `CORBA::LocalObject` to provide implementations of `Object` pseudo-operations, and other ORB-specific support mechanisms that apply. Because object implementations are language-specific, the `LocalObject` type is only defined by each language mapping.

**Local object pseudo-operations**

The `LocalObject` type implements the `Object` pseudo-operations shown in [Table 15](#).

**Table 15:** *CORBA::LocalObject Pseudo-Operations and Return Values*

Operation	Always returns...
<code>is_a()</code>	an exception of <code>NO_IMPLEMENT</code> .
<code>get_interface()</code>	an exception of <code>NO_IMPLEMENT</code> .
<code>get_domain_managers()</code>	an exception of <code>NO_IMPLEMENT</code> .
<code>get_policy()</code>	an exception of <code>NO_IMPLEMENT</code> .
<code>get_client_policy()</code>	an exception of <code>NO_IMPLEMENT</code> .
<code>set_policy_overrides()</code>	an exception of <code>NO_IMPLEMENT</code> .
<code>get_policy_overrides()</code>	an exception of <code>NO_IMPLEMENT</code> .
<code>validate_connection()</code>	an exception of <code>NO_IMPLEMENT</code> .
<code>non_existent()</code>	false.
<code>hash()</code>	a hash value that is consistent with the object's lifetime.
<code>is_equivalent()</code>	true, if the references refer to the same <code>LocalObject</code> implementation.

---

# Valuetypes

## Overview

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

**Note:** The COBOL runtime and the Orbix E2A IDL compiler do not support valuetypes.

## Characteristics

Valuetypes can be seen as a cross between the following:

- Data types, such as `long` and `string`, which can be passed by value over the wire as arguments to remote invocations.
- Objects, which can only be passed by reference.

When a program supplies an object reference, the object remains in its original location; subsequent invocations on that object from other address spaces move across the network, rather than the object moving to the site of each request.

## Valuetype support

Like an interface, a valuetype supports both operations and inheritance from other valuetypes; it also can have data members. When a valuetype is passed as an argument to a remote operation, the receiving address space creates a copy of it. The copied valuetype exists independently of the original; operations that are invoked on one have no effect on the other.

## Valuetype invocations

Because a valuetype is always passed by value, its operations can only be invoked locally. Unlike invocations on objects, valuetype invocations are never passed over the wire to a remote valuetype.

**Valuetype implementations**

---

Valuetype implementations necessarily vary, depending on the languages used on sending and receiving ends of the transmission, and their respective abilities to marshal and demarshal the valuetype's operations. A receiving process that is written in C++ must provide a class that implements valuetype operations and a factory to create instances of that class. These classes must be either compiled into the application, or made available through a shared library. Conversely, Java applications can marshal enough information on the sender, so the receiver can download the bytecodes for the valuetype operation implementations.

---

# 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 COBOL runtime and the Orbix E2A IDL compiler do not support abstract interfaces.

## IDL abstract interface sample

In the following example, the IDL definitions specify that the `Example::display()` operation accepts any derivation of the abstract interface, `Describable`:

```
abstract interface Describable {
    string get_description();
};

interface Example {
    void display(in Describable someObject);
};
```

## Abstract interface IDL sample

Based on the preceding IDL, you can define two derivations of the `Describable` abstract interface—the `Currency` valuetype and the `Account` interface:

```
interface Account : Describable {
    // body of Account definition not shown
};

valuetype Currency supports Describable {
    // body of Currency definition not shown
};
```

**Note:** Because the parameter for `display()` is defined as a `Describable` type, invocations on this operation can supply either `Account` objects or `Currency` valuetypes.

---

# IDL Data Types

**In this section**

The following topics are discussed in this section:

Built-in Data Types	page 141
Extended Built-in Data Types	page 143
Complex Data Types	page 146
Enum Data Type	page 147
Struct Data Type	page 148
Union Data Type	page 149
Arrays	page 151
Sequence	page 152
Pseudo Object Types	page 153

**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 COBOL or PL/I.

## Built-in Data Types

### List of types, sizes, and values

Table 16 shows a list of CORBA IDL built-in data types (where the  $\leq$  symbol means 'less than or equal to').

**Table 16:** *Built-in IDL Data Types, Sizes, and Values*

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

### 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
```

---

**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`. Refer to 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 17 shows a list of CORBA IDL extended built-in data types (where the  $\leq$  symbol means 'less than or equal to').

Table 17: *Extended built-in IDL Data Types, Sizes, and Values*

Data Type	Size	Range of Values
long long <sup>a</sup>	$\leq 64$ bits	$-2^{63} \dots 2^{63}-1$
unsigned long long <sup>a</sup>	$\leq$	

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 E2A 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 COBOL compiler for OS/390, only up to 18 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 (+/-)999999999.99.

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

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

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

### Constant fixed types

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

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

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

### Fixed type and decimal fractions

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

---

## Complex Data Types

---

### IDL complex data types

IDL provide the following complex data types:

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

---

## Enum Data Type

### Overview

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

### Enum IDL sample

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

```
module BankDemo {  
    enum Currency {pound, dollar, yen, franc};  
  
    interface Account {  
        readonly attribute CashAmount balance;  
        readonly attribute Currency balanceCurrency;  
        //...  
    };  
};
```

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

### Ordinal values of enum type

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

---

## Struct Data Type

### Overview

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

### Struct IDL sample

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

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

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

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

---

# Union Data Type

## Overview

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

## Union declaration syntax

You declare a union type with the following syntax:

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

## Discriminated unions

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

## IDL union date sample

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

```
enum dateStorage  
{ numeric, strMMDDYY, strDDMMYY };  
  
struct DateStructure {  
    short Day;  
    short Month;  
    short Year;  
};  
  
union Date switch (dateStorage) {  
    case numeric: long digitalFormat;  
    case strMMDDYY:  
    case strDDMMYY: string stringFormat;  
    default: DateStructure structFormat;  
};
```

**Sample explanation**

---

Given the preceding IDL:

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

**Rules for union types**

The following rules apply to union types:

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



---

# Arrays

## Overview

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

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

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

## Array IDL sample

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

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

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

## Array indexes

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

---

## Sequence

### Overview

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

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

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

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

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

### Bounded and unbounded sequences

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

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

### Bounded and unbounded IDL definitions

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

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

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

---

## Pseudo Object Types

---

### Overview

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

**Note:** The COBOL runtime and the Orbix E2A IDL compiler do not support all pseudo object types.

### Defining

You can use only the following pseudo-object types as attribute or operation parameter types in an IDL specification:

```
CORBA::NamedValue  
CORBA::TypeCode
```

To use these types in an IDL specification, include the `orb.idl` file in the IDL file as follows:

```
#include <orb.idl>  
//...
```

This statement instructs the IDL compiler to allow the `NamedValue` and `TypeCode` types.

---

# Defining Data Types

---

**In this section**

This section contains the following subsections:

<a href="#">Constants</a>	<a href="#">page 155</a>
<a href="#">Constant Expressions</a>	<a href="#">page 158</a>

---

**Using typedef**

With `typedef`, you can define more meaningful or simpler names for existing data types, regardless of whether those types are IDL-defined or user-defined.

---

**Typedef identifier IDL sample**

The following code defines the `typedef` identifier, `StandardAccount`, so that it can act as an alias for the `Account` type in later IDL definitions:

```
module BankDemo {  
    interface Account {  
        //...  
    };  
  
    typedef Account StandardAccount;  
};
```



**Example 10:** *List of character constants (Sheet 2 of 2)*

```

const char C11 = '\\';      // backslash
const char C12 = '\\?';    // question mark
const char C13 = '\\';     // single quote
// String constants support the same escape sequences as C++
const string S1 = "Quote: \""; // string with double quote
const string S2 = "hello world"; // simple string
const string S3 = "hello" " world"; // concatenate
const string S4 = "\\xA" "B";      // two characters
                                   // ('\\xA' and 'B'),
                                   // not the single character '\\xAB'

```

**Wide character and string constants**

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

```

const wchar C = L'X';
const wstring GREETING = L"Hello";
const wchar OMEGA = L'\\u03a9';
const wstring OMEGA_STR = L"Omega: \\u3A9";

```

IDL files always use the ISO Latin-1 code set; they cannot use Unicode or other extended character sets.

**Boolean constants**

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

```

// There is no need to define boolean constants:
const CONTRADICTION = FALSE; // Pointless and confusing
const TAUTOLOGY = TRUE;     // Pointless and confusing

```

**Octet constants**

Octet constants are positive integers in the range 0-255.

```

const octet O1 = 23;
const octet O2 = 0xf0;

```

Octet constants were added with CORBA 2.3; therefore, ORBs that are not compliant with this specification might not support them.

## Fixed-point constants

For fixed-point constants, you do not explicitly specify the digits and scale. Instead, they are inferred from the initializer. The initializer must end in `d` or `D`. For example:

```
// Fixed point constants take digits and scale from the
// initializer:
const fixed val1 = 3D;           // fixed<1,0>
const fixed val2 = 03.14d;      // fixed<3,2>
const fixed val3 = -03000.00D;  // fixed<4,0>
const fixed val4 = 0.03D;       // fixed<3,2>
```

The type of a fixed-point constant is determined after removing leading and trailing zeros. The remaining digits are counted to determine the digits and scale. The decimal point is optional.

Currently, there is no way to control the scale of a constant if it ends in trailing zeros.

## Enumeration constants

Enumeration constants must be initialized with the scoped or unscoped name of an enumerator that is a member of the type of the enumeration. For example:

```
enum Size { small, medium, large }

const Size DFL_SIZE = medium;
const Size MAX_SIZE = ::large;
```

Enumeration constants were added with CORBA 2.3; therefore, ORBs that are not compliant with this specification might not support them.

---

## Constant Expressions

### Overview

IDL provides a number of arithmetic and bitwise operators. The arithmetic operators have the usual meaning and apply to integral, floating-point, and fixed-point types (except for %, which requires integral operands). However, these operators do not support mixed-mode arithmetic: you cannot, for example, add an integral value to a floating-point value.

### Arithmetic operators

The following code contains several examples of arithmetic operators:

```
// You can use arithmetic expressions to define constants.
const long MIN = -10;
const long MAX = 30;
const long DFLT = (MIN + MAX) / 2;

// Can't use 2 here
const double TWICE_PI = 3.1415926 * 2.0;

// 5% discount
const fixed DISCOUNT = 0.05D;
const fixed PRICE = 99.99D;

// Can't use 1 here
const fixed NET_PRICE = PRICE * (1.0D - DISCOUNT);
```

### Evaluating expressions for arithmetic operators

Expressions are evaluated using the type promotion rules of C++. The result is coerced back into the target type. The behavior for overflow is undefined, so do not rely on it. Fixed-point expressions are evaluated internally with 31 bits of precision, and results are truncated to 15 digits.



---

**Bitwise operators**

Bitwise operators only apply to integral types. The right-hand operand must be in the range 0-63. The right-shift operator, `>>`, is guaranteed to insert zeros on the left, regardless of whether the left-hand operand is signed or unsigned.

```
// You can use bitwise operators to define constants.  
const long ALL_ONES = -1;           // 0xffffffff  
const long LHW_MASK = ALL_ONES << 16; // 0xffff0000  
const long RHW_MASK = ALL_ONES >> 16; // 0x0000ffff
```

IDL guarantees two's complement binary representation of values.

---

**Precedence**

The precedence for operators follows the rules for C++. You can override the default precedence by adding parentheses.



# IDL-to-COBOL 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-COBOL mapping rules and shows, by example, how each IDL type is represented in COBOL.*

**In this chapter**

This chapter discusses the following topics:

Mapping for Identifier Names	page 163
Mapping for Type Names	page 165
Mapping for Basic Types	page 166
Mapping for Boolean Type	page 171
Mapping for Enum Type	page 174
Mapping for Char Type	page 177
Mapping for Octet Type	page 179
Mapping for String Types	page 181
Mapping for Wide String Types	page 186

Mapping for Fixed Type	page 187
Mapping for Struct Type	page 191
Mapping for Union Type	page 193
Mapping for Sequence Types	page 198
Mapping for Array Type	page 203
Mapping for the Any Type	page 205
Mapping for User Exception Type	page 207
Mapping for Typedefs	page 210
Mapping for the Object Type	page 213
Mapping for Constant Types	page 215
Mapping for Operations	page 218
Mapping for Attributes	page 223
Mapping for Operations with a Void Return Type and No Parameters	page 224
Mapping for Inherited Interfaces	page 226
Mapping for Multiple Interfaces	page 233

**Note:** See “IDL Interfaces” on page 117 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 COBOL.

## COBOL rules for identifiers

The following rules apply for COBOL identifiers:

- They can be a maximum of 30 characters in length.
- They can only consist of alphanumeric and hyphen characters.

## IDL-to-COBOL mapping rules for identifiers

The following rules are used to convert an IDL identifier to a COBOL identifier:

- Replace each underscore with a hyphen.
- Remove any leading or trailing hyphens.
- If an identifier clashes with a reserved COBOL word, prefix it with the characters `IDL-`. For example, `d` maps to `IDL-D`, `u` maps to `IDL-U`, and `result` maps to `IDL-RESULT`.
- If an identifier is greater than 30 characters, truncate it to 30 characters, by using the first 25 characters followed by a hyphen followed by a unique alphanumeric four-character suffix.

## Example

The example can be broken down as follows:

1. Consider the following IDL:

```
module amodule
{
    interface example
    {
        attribute boolean myverylongattribute;
        boolean myverylongopname(in boolean
            myverylongboolean);
    };
};
```

2. The preceding IDL maps to the following COBOL:

```
*****
* Interface:
*   amodule/example
*
* Mapped name:
*   amodule-example
*
* Inherits interfaces:
*   (none)
*****
*****
* Attribute:   myverylongattribute
* Mapped name: myverylongattribute
* Type:       boolean (read/write)
*****
01 AMODULE-EXAMPLE-MYVE-5905-ARGS.
   03 RESULT                                     PICTURE 9(01)
                                           BINARY.
           88 RESULT-FALSE                     VALUE 0.
           88 RESULT-TRUE                       VALUE 1.
*****
* Operation:   myverylongopname
* Mapped name: myverylongopname
* Arguments:   <in> boolean myverylongboolean
* Returns:    boolean
* User Exceptions: none
*****
01 AMODULE-EXAMPLE-MYVE-EAB7-ARGS.
   03 MYVERYLONGBOOLEAN                         PICTURE 9(01)
                                           BINARY.
           88 MYVERYLONGBOOLEAN-FALSE           VALUE 0.
           88 MYVERYLONGBOOLEAN-TRUE            VALUE 1.
   03 RESULT                                     PICTURE 9(01)
                                           BINARY.
           88 RESULT-FALSE                     VALUE 0.
           88 RESULT-TRUE                       VALUE 1.
```

**Note:** See “-M Argument” on page 244 and “-O Argument” on page 250 for details of the arguments that you can use with the Orbix E2A IDL compiler to create alternative COBOL identifiers.

---

# Mapping for Type Names

---

## Overview

This section describes how IDL type names are mapped to COBOL.

---

## IDL-to-COBOL mapping for type names

The current CORBA OMG COBOL mapping is based on the use of typedefs for naming some IDL types. Typedefs are a non-standard extension to the COBOL-85 standard. The IBM COBOL compiler for OS/390 & VM version 2 release 1 does not support this extension.

The CORBA COBOL mapping standard includes a recent addition that proposes the use of `COPY ... REPLACING` syntax instead of typedefs for type definitions. IONA currently uses the COBOL representation of each type directly.

# Mapping for Basic Types

## Overview

This section describes how basic IDL types are mapped to COBOL.

## IDL-to-COBOL mapping for basic types

Table 18 shows the mapping rules for basic IDL types. Types not currently supported by Orbix COBOL are denoted by *italic* text. The CORBA typedef name is provided for reference purposes only; the COBOL representation is used directly.

**Table 18:** *Mapping for Basic IDL Types (Sheet 1 of 2)*

IDL Type	CORBA Typedef Name	COBOL Representation
short	CORBA-short	PIC S9(05) BINARY
long	CORBA-long	PIC S9(10) BINARY
unsigned short	CORBA-unsigned-short	PIC 9(05) BINARY
unsigned long	CORBA-unsigned-long	PIC 9(10) BINARY
float	CORBA-float	COMP-1
double	CORBA-double	COMP-2
char	CORBA-char	PIC X
boolean	CORBA-boolean	PIC 9(01) BINARY
octet	CORBA-octet	PIC X
enum	CORBA-enum	PIC 9(10) BINARY
fixed<d,s>	Fixed<d,s>	PIC S9(d-s)v(s) PACKED-DECIMAL
fixed<d,-s>	Fixed<d,-s>	PIC S9(d)P(s) PACKED-DECIMAL



**Table 18:** *Mapping for Basic IDL Types (Sheet 2 of 2)*

IDL Type	CORBA Typedef Name	COBOL Representation
any	CORBA-any	Refer to <a href="#">“Mapping for the Any Type”</a> on page 205.
<i>long long</i>	<i>CORBA-long-long</i>	<i>PIC S9(18) BINARY</i>
<i>unsigned long long</i>	<i>CORBA-unsigned-long-long</i>	<i>PIC 9(18) BINARY</i>
<i>wchar</i>	<i>CORBA-wchar</i>	<i>PIC G</i>

**Example**

The example can be broken down as follows:

1. Consider the following IDL:

```
const float my_outer_float = 19.76;
const double my_outer_double = 123456.789;

interface example
{
    const short my_short = 24;
    const long my_long = 9999;
    typedef fixed<5,2> a_fixed_5_2;
    attribute short myshort;
    attribute long mylong;
    attribute unsigned short myushort;
    attribute unsigned long myulong;
    attribute float myfloat;
    attribute double mydouble;
    attribute char mychar;
    attribute octet myoctet;
    attribute a_fixed_5_2 myfixed_5_2;
    attribute long long mylonglong;
    attribute unsigned long long ulonglong;
};
```

2. The preceding IDL maps to the following COBOL:

**Example 11:** *COBOL Example for Basic Types (Sheet 1 of 3)*

```
*****
* Constants in root scope:
*****
01 GLOBAL-EXAM1A-CONSTS.
    03 MY-OUTER-FLOAT                COMPUTATIONAL-1
                                      VALUE 1.976e+01.
    03 MY-OUTER-DOUBLE              COMPUTATIONAL-2
                                      VALUE 1.23456789e+05.
*****
* Interface:
*   example
*
* Mapped name:
*   example
*
* Inherits interfaces:
*   (none)
*****
* Attribute:    myshort
* Mapped name:  myshort
* Type:         short (read/write)
*****
01 EXAMPLE-MYSHORT-ARGS.
    03 RESULT                PICTURE S9(05)
                              BINARY.
*****
* Attribute:    mylong
* Mapped name:  mylong
* Type:         long (read/write)
*****
01 EXAMPLE-MYLONG-ARGS.
    03 RESULT                PICTURE S9(10)
                              BINARY.
*****
* Attribute:    myushort
* Mapped name:  myushort
* Type:         unsigned short (read/write)
*****
01 EXAMPLE-MYUSHORT-ARGS.
    03 RESULT                PICTURE 9(05)
                              BINARY.
```

**Example 11:** *COBOL Example for Basic Types (Sheet 2 of 3)*

```

*****
* Attribute:      myulong
* Mapped name:    myulong
* Type:           unsigned long (read/write)
*****
01 EXAMPLE-MYULONG-ARGS.
   03 RESULT                                PICTURE 9(10)
                                           BINARY.
*****
* Attribute:      myfloat
* Mapped name:    myfloat
* Type:           float (read/write)
*****
01 EXAMPLE-MYFLOAT-ARGS.
   03 RESULT                                COMPUTATIONAL-1.
*****
* Attribute:      mydouble
* Mapped name:    mydouble
* Type:           double (read/write)
*****
01 EXAMPLE-MYDOUBLE-ARGS.
   03 RESULT                                COMPUTATIONAL-2.
*****
* Attribute:      mychar
* Mapped name:    mychar
* Type:           char (read/write)
*****
01 EXAMPLE-MYCHAR-ARGS.
   03 RESULT                                PICTURE X(01).
*****
* Attribute:      myoctet
* Mapped name:    myoctet
* Type:           octet (read/write)
*****
01 EXAMPLE-MYOCTET-ARGS.
   03 RESULT                                PICTURE X(01).
*****
* Attribute:      myfixed_5_2
* Mapped name:    myfixed_5_2
* Type:           example/a_fixed_5_2 (read/write)
*****
01 EXAMPLE-MYFIXED-5-2-ARGS.
   03 RESULT                                PICTURE S9(3)V9(2)
                                           PACKED-DECIMAL.

```

**Example 11:** *COBOL Example for Basic Types (Sheet 3 of 3)*

```

*****
* Attribute:      mylonglong
* Mapped name:    mylonglong
* Type:           long long (read/write)
*****
01 EXAMPLE-MYLONGLONG-ARGS.
   03 RESULT                                     PICTURE S9(18)
                                           BINARY.
*****
* Attribute:      ulonglong
* Mapped name:    ulonglong
* Type:           unsigned long long (read/write)
*****
01 EXAMPLE-ULONGLONG-ARGS.
   03 RESULT                                     PICTURE 9(18)
                                           BINARY.
*****
* Constants in example:
*****
01 EXAMPLE-CONSTS.
   03 MY-SHORT                                     PICTURE S9(05)
                                           BINARY VALUE 24.
   03 MY-LONG                                     PICTURE S9(10)
                                           BINARY VALUE 9999.

```

---

# Mapping for Boolean Type

---

## Overview

This section describes how booleans are mapped to COBOL.

## IDL-to-COBOL mapping for booleans

An IDL boolean type maps to a COBOL `PIC 9(01)` integer value and has two COBOL conditions defined, as follows:

- A label `idl-identifier-FALSE` with a 0 value.
- A label `idl-identifier-TRUE` with a 1 value.

**Note:** The IBM COBOL compiler for OS/390 & VM does not currently support the non-COBOL85 `>>CONSTANT` construct. This is specified for the mapping of constant boolean values. Responsibility is passed to the Orbix E2A IDL compiler to propagate constant values. In this case, the following mapping approach that uses `Level 88` items has been chosen:

## Example

The example can be broken down as follows:

1. Consider the following IDL, which is contained in an IDL member called `EXAM1`:

```
// IDL
interface example {
    attribute boolean full;
    boolean myop(in boolean myboolean);
}
```

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following COBOL in the `EXAM1` copybook:

```
*****
* Attribute:      full
* Mapped name:    full
* Type:           boolean (read/write)
*****
01 EXAMPLE-FULL-ARGS.
   03 RESULT                                PICTURE 9(01) BINARY.
       88 RESULT-FALSE                      VALUE 0.
       88 RESULT-TRUE                       VALUE 1.
*****
* Operation:      myop
* Mapped name:    myop
* Arguments:       <in> boolean myboolean
* Returns:         boolean
* User Exceptions: none
*****
01 EXAMPLE-MYOP-ARGS.
   03 MYBOOLEAN                                PICTURE 9(01) BINARY.
       88 MYBOOLEAN-FALSE                    VALUE 0.
       88 MYBOOLEAN-TRUE                    VALUE 1.
   03 RESULT                                PICTURE 9(01) BINARY.
       88 RESULT-FALSE                      VALUE 0.
       88 RESULT-TRUE                      VALUE 1.
01 EXAMPLE-OPERATION                                PICTURE X(26).
   88 EXAMPLE-GET-FULL                            VALUE
       "_get_full:IDL:example:1.0".
   88 EXAMPLE-SET-FULL                            VALUE
       "_set_full:IDL:example:1.0".
   88 EXAMPLE-MYOP                                VALUE
       "myop:IDL:example:1.0".
01 EXAMPLE-OPERATION-LENGTH                        PICTURE 9(09) BINARY
       VALUE 26.
```

3. The preceding code can be used as follows:

```
IF RESULT-TRUE OF RESULT OF EXAMPLE-FULL-ARGS THEN
    SET EXAMPLE-SET-FULL TO TRUE
ELSE
    SET EXAMPLE-GET-FULL TO TRUE
END-IF
CALL "ORBEXEC" USING SERVER-OBJ
    EXAMPLE-OPERATION
    EXAMPLE-FULL-ARGS
    EXAM1-USER-EXCEPTIONS
```

---

# Mapping for Enum Type

---

## Overview

This section describes how enums are mapped to COBOL.

---

## IDL-to-COBOL mapping for enums

An IDL enum type maps to a COBOL `PIC 9(10) BINARY` type. The COBOL mapping for an enum is an unsigned integer capable of representing  $2^{32}$  enumerations (that is,  $2^{32-1}$  enumerations). Because IDL does not allow you to set ordinal values for enums, each identifier in a mapped enum has a COBOL condition defined with its own appropriate integer value, based on the rule that integer values are incrementing and start at 0. Each identifier is a level 88 entry.

---

## Example

The example can be broken down as follows:

1. Consider the following IDL, which is contained in an IDL member called `EXAM2`:

```
// IDL
interface example {
    enum temp {cold, warm, hot };
    attribute temp attr1;
    temp myop(in temp myenum);
}
```



2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following COBOL in the `EXAM2` copybook:

```
*****
* Attribute:      attr1
* Mapped name:    attr1
* Type:           temp (read/write)
*****
01 EXAMPLE-ATTR1-ARGS.
    03 RESULT                                PICTURE 9(10) BINARY.
        88 COLD                                VALUE 0.
        88 WARM                                VALUE 1.
        88 HOT                                 VALUE 2.
*****
* Operation:      myop
* Mapped name:    myop
* Arguments:      <in> temp myenum
* Returns:        temp
* User Exceptions: none
*****
01 EXAMPLE-MYOP-ARGS.
    03 MYENUM                                PICTURE 9(10) BINARY.
        88 COLD                                VALUE 0.
        88 WARM                                VALUE 1.
        88 HOT                                 VALUE 2.
    03 RESULT                                PICTURE 9(10) BINARY.
        88 COLD                                VALUE 0.
        88 WARM                                VALUE 1.
        88 HOT                                 VALUE 2.
```

3. The preceding code can be used as follows:

```
EVALUATE TRUE
  WHEN COLD OF EXAMPLE-ATTR1-ARGS
  ...
  WHEN WARM OF EXAMPLE-ATTR1-ARGS
  ...
  WHEN HOT OF EXAMPLE-ATTR1-ARGS
  ...
END-EVALUATE
```

---

# Mapping for Char Type

---

## Overview

This section describes how char types are mapped to COBOL.

---

## IDL-to-COBOL 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 translated by the ORB.

---

## Example

The example can be broken down as follows:

1. Consider the following IDL, which is contained in an IDL member called `EXAM3`:

```
// IDL
interface example {
    attribute char achar;
    char myop(in char mychar);
}
```

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following COBOL in the `EXAM3` copybook:

```
*****
* Attribute:      achar
* Mapped name:    achar
* Type:           char (read/write)
*****
01 EXAMPLE-ACHAR-ARGS.
   03 RESULT                                PICTURE X(01).
*****
* Operation:      myop
* Mapped name:    myop
* Arguments:      <in> char mychar
* Returns:        char
* User Exceptions: none
*****
01 EXAMPLE-MYOP-ARGS.
   03 MYCHAR                                PICTURE X(01).
   03 RESULT                                PICTURE X(01).
```

---

# Mapping for Octet Type

---

## Overview

This section describes how octet types are mapped to COBOL.

---

## IDL-to-COBOL mapping for octet types

The octet type refers to binary character data. The ORB does not translate any octet data, even if the remote system has a different character set than the local system (for example ASCII and EBCDIC). You should take special care in selecting the appropriate IDL type when representing text data (that is, a string) as opposed to IDL 58 `binary` (that is, an octet).

---

## Example

The example can be broken down as follows:

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following COBOL in the `EXAM4` copybook:

```
*****
* Attribute:      aoctet
* Mapped name:    aoctet
* Type:           octet (read/write)
*****
01 EXAMPLE-AOCTET-ARGS.
   03 RESULT                                PICTURE X(01).
*****
* Operation:      myop
* Mapped name:    myop
* Arguments:      <in> char myoctet
* Returns:        octet
* User Exceptions: none
*****
01 EXAMPLE-MYOP-ARGS.
   03 MYOCTET                                PICTURE X(01).
   03 RESULT                                PICTURE X(01).
```

---

# Mapping for String Types

---

## Overview

This section describes how string types are mapped to COBOL. First, it describes the various string types that are available.

---

## Bounded and unbounded strings

Strings can be bounded or unbounded. Bounded strings are of a specified size, while unbounded strings have no specified size. For example:

```
//IDL
string<8>  a_bounded_string
string     an_unbounded_string
```

Bounded and unbounded strings are represented differently in COBOL.

---

## Incoming bounded strings

Incoming strings are passed as `IN` or `INOUT` values by the `COAGET` function into the COBOL operation parameter buffer at the start of a COBOL operation.

An incoming bounded string is represented by a COBOL `PIC X(n)` data item, where  $n$  is the bounded length of the string. For example:

1. Consider the following IDL:

```
interface example {
    typedef string<10> boundedstr;
    attribute boundedstr aboundedstr;
    boundedstr myop(in boundedstr myboundedstr);
};
```

2. The preceding IDL maps to the following COBOL:

```
*****
* Attribute:      aboundedstr
* Mapped name:    aboundedstr
* Type:           example/boundedstr (read/write)
*****
01 EXAMPLE-ABOUNDEDSTR-ARGS.
   03 RESULT                                           PICTURE X(10).
*****
* Operation:      myop
* Mapped name:    myop
* Arguments:       <in> example/boundedstr myboundedstr
* Returns:        example/boundedstr
* User Exceptions: none
*****
01 EXAMPLE-MYOP-ARGS.
   03 MYBOUNDEDSTR                                     PICTURE X(10).
   03 RESULT                                           PICTURE X(10).
*****
```

If the string that is passed is too big for the buffer, the string is truncated. If the string is not big enough to fill the buffer, the remainder of the COBOL string is filled with spaces.

### Outgoing bounded strings

Outgoing strings are copied as `INOUT`, `OUT`, or `RESULT` values by the `COAPUT` function from the complete COBOL operation parameter buffer that is passed to it at the end of a COBOL operation.

An outgoing bounded string has trailing spaces removed, and all characters up to the bounded length (or the first null) are passed via `COAPUT`. 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 `COAGET` function into the COBOL operation parameter buffer at the start of a COBOL operation.

An incoming unbounded string is represented as a `USAGE IS POINTER` data item. For example:

1. Consider the following IDL:

```
interface example {
    typedef string unboundedstr;
    attribute unboundedstr aunboundedstr;
    unboundedstr myop(in unboundedstr myunboundedstr);
};
```

2. The preceding IDL maps to the following COBOL:

```
*****
* Attribute:      aunboundedstr
* Mapped name:    aunboundedstr
* Type:           example/unboundedstr (read/write)
*****
01 EXAMPLE-AUNBOUNDEDSTR-ARGS.
   03 RESULT                                POINTER VALUE NULL.
*****
* Operation:      myop
* Mapped name:    myop
* Arguments:      <in> example/unboundedstr munyboundedstr
* Returns:        example/unboundedstr
* User Exceptions: none
*****
01 EXAMPLE-MYOP-ARGS.
   03 MUNYBOUNDEDSTR                        POINTER VALUE NULL.
   03 RESULT                                POINTER VALUE NULL.
```

3. A pointer is supplied which 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 COBOL `PIC X(n)` structure. For example:

```
* This is the supplied COBOL unbounded string pointer

01 NAME                                USAGE IS POINTER

* This is the COBOL representation of the string

01 SUPPLIER-NAME                       PICTURE X(64).
01 SUPPLIER-NAME-LEN                   PICTURE 9(10) BINARY VALUE 64.

* This STRGET call copies the characters in the NAME
* to the SUPPLIER-NAME

CALL "STRGET"                          USING NAME
                                         SUPPLIER-NAME-LEN
                                         SUPPLIER-NAME.
```

In the preceding example, the number of characters copied depends on the value specified for `SUPPLIER-NAME-LEN`. This must be a valid positive integer (that is, greater than zero); otherwise, a runtime error occurs. If the value specified for `SUPPLIER-NAME` is shorter than that for `SUPPLIER-NAME-LEN`, the string is still copied to `SUPPLIER-NAME`, but it obviously cannot contain the complete string.

### Outgoing unbounded strings

Outgoing strings are copied as `INOUT`, `OUT`, or `RESULT` values by the `COAPUT` function from the complete COBOL operation parameter buffer that is passed to it at the end of a COBOL operation.

A valid outgoing unbounded string must be supplied by the implementation of an operation. This can be either a pointer that was obtained by an `IN` or `INOUT` parameter, or a string constructed by using the `STRSET` function. For example:

```
* This is the COBOL representation of the string containing a
* value that we want to pass back to the client using COAPUT
* via an unbounded pointer string.                                */

01 NOTES                                PICTURE X(160).
01 NOTES-LEN                            PICTURE 9(10) BINARY
                                         VALUE 160.

* This is the unbounded pointer string

01 CUST-NOTES                           USAGE IS POINTER.

* This STRSET call creates an unbounded string called CUST-NOTES
* to which it copies NOTES-LEN characters from character string
* NOTES

CALL "STRSET"                           USING CUST-NOTES
                                         NOTES-LEN
                                         NOTES.
```

Trailing spaces are removed from the constructed string. If trailing spaces are required, you can use the `STRSETP` function, with the same argument signature, to copy the specified number of characters, including trailing spaces.

---

# Mapping for Wide String Types

---

## Overview

This section describes how wide string types are mapped to COBOL.

---

## IDL-to-COBOL mapping for wide strings

The mapping for the `wstring` type is similar to the mapping for strings, but it requires DBCS support from the IBM COBOL compiler for OS/390 & VM. The current IBM COBOL compiler for OS/390 & VM does have DBCS support.

A `PICTURE G` (instead of a `PICTURE X`) data item represents the COBOL data item. Instead of calling `STRGET` and `STRSET` to access unbounded strings, the auxiliary functions `WSTRGET` and `WSTRSET` should be used. The argument signatures for these functions are equivalent to their string counterparts.

---

# Mapping for Fixed Type

## Overview

This section describes how fixed types are mapped to COBOL.

## IDL-to-COBOL mapping for fixed types

The IDL fixed type maps directly to COBOL packed decimal data with the appropriate number of digits and decimal places (if any).

**Note:** All fixed types must be declared in IDL with `typedef`.

## The 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:

```
typedef fixed<9,2> net_worth; // up to $9,999,999.99, accurate to
                             // one cent.
typedef fixed<9,4> exchange_rate; // accurate to 1/10000 unit.
typedef fixed<9,0> annual_revenue; // in millions
typedef fixed<3,6> wrong; // this is invalid.
```

## 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-COBOL mapping for fixed types

The example can be broken down as follows:

1. Consider the following IDL:

```
//IDL
interface example
{
    typedef fixed<10,0> type_revenue;
    attribute type_revenue revenue;
    typedef fixed<6,4> type_precise;
    attribute type_precise precise;
    type_precise myop(in type_revenue myfixed);
    typedef fixed<6,-4> type_millions;
    attribute type_millions millions;
};
```

2. The preceding IDL maps to the following COBOL:

### Example 12: COBOL Example for Fixed Type (Sheet 1 of 2)

```
*****
* Attribute:    revenue
* Mapped name:  revenue
* Type:         example/type_revenue (read/write)
*****
01 EXAMPLE-REVENUE-ARGS.
   03 RESULT                                PICTURE S9(10)
                                           PACKED-DECIMAL.
*****
* Attribute:    precise
* Mapped name:  precise
* Type:         example/type_precise (read/write)
*****
01 EXAMPLE-PRECISE-ARGS.
   03 RESULT                                PICTURE S9(2)V9(4)
                                           PACKED-DECIMAL.
*****
* Attribute:    millions
* Mapped name:  millions
* Type:         example/type_millions (read/write)
*****
01 EXAMPLE-MILLIONS-ARGS.
   03 RESULT                                PICTURE S9(6)P(4)
                                           PACKED-DECIMAL.
*****
* Operation:    myop
```

**Example 12: COBOL Example for Fixed Type (Sheet 2 of 2)**

```

* Mapped name:      myop
* Arguments:        <in> example/type_revenue myfixed
* Returns:          example/type_precise
* User Exceptions:  none
*****
01 EXAMPLE-MYOP-ARGS.
   03 MYFIXED                                PICTURE S9(10)
                                           PACKED-DECIMAL.
   03 RESULT                                PICTURE S9(2)V9(4)
                                           PACKED-DECIMAL.

```

**Limitations in size of COBOL numeric data items**

The IBM COBOL compiler for OS/390 & VM version 2 release 1 limits numeric data items to a maximum of 18 digits, whereas the IDL fixed type specifies support for up to 31 digits. If the IDL definition specifies more than 18 digits, the generated data item is restricted to 18 digits. Truncation of the excess most-significant digits occurs when the item is passed to COBOL. Passing data from COBOL to a fixed type with greater than 18 digits results in zero-filling of the excess most-significant digits.

For example, consider the following IDL:

```

// IDL
interface example
{
    typedef fixed<25,0> lots_of_digits;
    attribute lots_of_digits large_value;

    typedef fixed<25,8> lots_of_digits_and_prec;
    attribute lots_of_digits_and_prec large_value_prec;
};

```

The preceding IDL cannot be represented in COBOL, because COBOL has a restricted maximum of 18 digits. The Orbix E2A IDL compiler issues a warning message and truncates to provide the following mapping:

```
*****
* Attribute:    large_value
* Mapped name:  large_value
* Type:         example/lots_of_digits (read/write)
*****
01 EXAMPLE-LARGE-VALUE-ARGS.
   03 RESULT                                         PICTURE S9(18)
                                                    PACKED-DECIMAL.
*****
* Attribute:    large_value_prec
* Mapped name:  large_value_prec
* Type:         example/lots_of_digits_and_prec (read/write)
*****
01 EXAMPLE-LARGE-VALUE-PREC-ARGS.
   03 RESULT                                         PICTURE S9(17)V9(1)
                                                    PACKED-DECIMAL.
```



---

# Mapping for Struct Type

---

## Overview

This section describes how struct types are mapped to COBOL.

---

## IDL-to-COBOL mapping for struct types

An IDL struct definition maps directly to COBOL group items.

---

## Example of IDL-to-COBOL mapping for struct types

The example can be broken down as follows:

1. Consider the following IDL:

```
// IDL
interface example
{
    struct a_structure
    {
        long      member1;
        short     member2;
        boolean    member3;
        string<10> member4;
    };
    typedef a_structure type_struct;
    attribute type_struct astruct;
    type_struct myop(in type_struct mystruct);
};
```

2. The preceding IDL maps to the following COBOL:

```
*****
* Attribute:      astruct
* Mapped name:    astruct
* Type:           example/type_struct (read/write)
*****
01 EXAMPLE-ASTRUCT-ARGS.
   03 RESULT.
      05 MEMBER1                PICTURE S9(10) BINARY.
      05 MEMBER2                PICTURE S9(05) BINARY.
      05 MEMBER3                PICTURE 9(01) BINARY.
         88 MEMBER3-FALSE        VALUE 0.
         88 MEMBER3-TRUE        VALUE 1.
      05 MEMBER4                PICTURE X(10).
*****
* Operation:      myop
* Mapped name:    myop
* Arguments:       <in> example/type_struct mystruct
* Returns:         example/type_struct
* User Exceptions: none
*****
01 EXAMPLE-MYOP-ARGS.
   03 MYSTRUCT.
      05 MEMBER1                PICTURE S9(10) BINARY.
      05 MEMBER2                PICTURE S9(05) BINARY.
      05 MEMBER3                PICTURE 9(01) BINARY.
         88 MEMBER3-FALSE        VALUE 0.
         88 MEMBER3-TRUE        VALUE 1.
      05 MEMBER4                PICTURE X(10).
   03 RESULT.
      05 MEMBER1                PICTURE S9(10) BINARY.
      05 MEMBER2                PICTURE S9(05) BINARY.
      05 MEMBER3                PICTURE 9(01) BINARY.
         88 MEMBER3-FALSE        VALUE 0.
         88 MEMBER3-TRUE        VALUE 1.
      05 MEMBER4                PICTURE X(10).
```

# Mapping for Union Type

## Overview

This section describes how union types are mapped to COBOL.

## IDL-to-COBOL mapping for union types

An IDL union definition maps directly to COBOL group items with the `REDEFINES` clause.

## Simple example of IDL-to-COBOL mapping for union types

The example can be broken down as follows:

1. Consider the following IDL:

```
// IDL
interface example
{
    union a_union switch(long)
    {
        case 1: char case_1;
        case 3: long case_3;
        default: string case_def;
    };
    typedef a_union type_union;
    attribute type_union aunion;
    type_union myop(in type_union myunion);
};
```

2. The preceding IDL maps to the following COBOL:

### Example 13: COBOL Example for Union Type (Sheet 1 of 2)

```
*****
* Attribute:      aunion
* Mapped name:    aunion
* Type:           example/type_union (read/write)
*****
01 EXAMPLE-AUNION-ARGS.
   03 RESULT.
      05 D                                     PICTURE S9(10) BINARY.
      05 U.
         07 FILLER                             PICTURE X(08)
                                                VALUE LOW-VALUES.
      05 FILLER REDEFINES U.
```

**Example 13:** *COBOL Example for Union Type (Sheet 2 of 2)*

```

07 CASE-1                                PICTURE X(01).
05 FILLER REDEFINES U.
07 CASE-3                                PICTURE S9(10) BINARY.
05 FILLER REDEFINES U.
07 CASE-DEF                              POINTER.
*****
* Operation:      myop
* Mapped name:    myop
* Arguments:      <in> example/type_union myunion
* Returns:        example/type_union
* User Exceptions: none
*****
01 EXAMPLE-MYOP-ARGS.
03 MYUNION.
05 D                                PICTURE S9(10) BINARY.
05 U.
07 FILLER                                PICTURE X(08)
VALUE LOW-VALUES.

05 FILLER REDEFINES U.
07 CASE-1                                PICTURE X(01).
05 FILLER REDEFINES U.
07 CASE-3                                PICTURE S9(10) BINARY.
05 FILLER REDEFINES U.
07 CASE-DEF                              POINTER.
03 RESULT.
05 D                                PICTURE S9(10) BINARY.
05 U.
07 FILLER                                PICTURE X(08)
VALUE LOW-VALUES.

05 FILLER REDEFINES U.
07 CASE-1                                PICTURE X(01).
05 FILLER REDEFINES U.
07 CASE-3                                PICTURE S9(10) BINARY.
05 FILLER REDEFINES U.
07 CASE-DEF                              POINTER.

```

### COBOL rules for mapped IDL unions

The following rules apply in COBOL for union types mapped from IDL:

1. The union discriminator in the group item is always referred to as `D`.
2. The union items are contained within the group item referred to as `U`.
3. Reference to union elements is made through the `EVALUATE` statement to test the discriminator.

**Note:** If `D` and `U` are used as IDL identifiers, they are treated as reserved words. This means that they are prefixed with `IDL-` in the generated COBOL (for example, the IDL identifier `d` maps to the COBOL identifier `IDL-D`).

### Example of COBOL rules for mapped IDL unions

The following code shows the COBOL rules for mapped IDL unions in effect:

```
EVALUATE D OF RESULT OF EXAMPLE-AUNION-ARGS
WHEN 1
  DISPLAY "its a character value = " CASE-1 OF U OF
    EXAMPLE-AUNION-ARGS
...
WHEN 3
  DISPLAY "its a long value = " CASE-3 OF U OF
    EXAMPLE-AUNION-ARGS
WHEN OTHER
  DISPLAY "its an unbounded string "
  * use strget to retrieve value
END-EVALUATE
```

### More complex example

The following provides a more complex example of the IDL-to-COBOL mapping rules for union types. The example can be broken down as follows:

1. Consider the following IDL:

```
interface example
{
    union a_union switch(long)
    {
        case 1: char case_1;
        case 3: long case_3;
        default: string case_def;
    };
    typedef a_union type_union;

    union a_nest_union switch(char)
    {
        case 'a': char case_a;
        case 'b': long case_b;
        case 'c': type_union case_c;
        default: string case_other;
    };
    typedef a_nest_union type_nest_union;

    attribute type_nest_union anestunion;
};
```

2. The preceding IDL maps to the following COBOL:

```
*****
* Attribute:      anestunion
* Mapped name:    anestunion
* Type:           example/type_nest_union (read/write)
*****
01 EXAMPLE-ANESTUNION-ARGS.
   03 RESULT.
      05 D                                     PICTURE X(01).
      05 U.
         07 FILLER                             PICTURE X(16)
                                         VALUE LOW-VALUES.

      05 FILLER REDEFINES U.
         07 CASE-A                             PICTURE X(01).

      05 FILLER REDEFINES U.
         07 CASE-B                             PICTURE S9(10) BINARY.
         05 FILLER REDEFINES U.
            07 CASE-C.
               09 D-1                         PICTURE S9(10) BINARY.
               09 U-1.
                  11 FILLER                     PICTURE X(08).
               09 FILLER REDEFINES U-1.
                  11 CASE-1                     PICTURE X(01).
               09 FILLER REDEFINES U-1.
                  11 CASE-3                     PICTURE S9(10) BINARY.
               09 FILLER REDEFINES U-1.
                  11 CASE-DEF                     POINTER.
      05 FILLER REDEFINES U.
         07 CASE-OTHER                         POINTER.
```

---

# Mapping for Sequence Types

Overview	This section describes how sequence types are mapped to COBOL. First, it describes the various sequence types that are available.
Bounded and unbounded sequences	<p>A sequence can be either bounded or unbounded. A bounded sequence is of a specified size, while an unbounded sequence has no specified size. For example:</p> <pre>// IDL typedef sequence&lt;long,10&gt; bounded seq attribute boundedseq seq1 typedef sequence&lt;long&gt; unboundedseq attribute unboundedseq seq2</pre> <p>Bounded and unbounded sequences are represented differently in COBOL. However, regardless of whether a sequence is bounded or unbounded, a supporting group item is always generated by the Orbix E2A IDL compiler, to provide some information about the sequence, such as the maximum length, the length of the sequence in elements, 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.</p>
Incoming and outgoing sequences	<p>A sequence that is being passed as an incoming parameter to a COBOL operation is passed as an <code>IN</code> or <code>INOUT</code> value by the <code>COGET</code> function into the operation parameter buffer at the start of the operation.</p> <p>A sequence that is being passed as an outgoing parameter or result from a COBOL operation is copied as an <code>INOUT</code>, <code>OUT</code>, or <code>RESULT</code> value by the <code>COAPUT</code> function from the complete operation parameter buffer that is passed to it at the end of the operation.</p>



## IDL-to-COBOL mapping for bounded sequences

A bounded sequence is represented by a COBOL `OCCURS` clause and a supporting group item. For example:

1. Consider the following IDL:

```
// IDL
interface example
{
    typedef sequence<long,10> boundedseq;
    attribute boundedseq aseq;
    boundedseq myop(in boundedseq myseq);
};
```

2. The preceding IDL maps to the following COBOL:

### Example 14: COBOL Example for Bounded Sequences (Sheet 1 of 2)

```
*****
* Attribute:      aseq
* Mapped name:    aseq
* Type:           example/boundedseq (read/write)
*****
01 EXAMPLE-ASEQ-ARGS.
   03 RESULT-1                                OCCURS 10 TIMES.
       05 RESULT                                PICTURE S9(10) BINARY.
   03 RESULT-SEQUENCE.
       05 SEQUENCE-MAXIMUM                    PICTURE 9(09) BINARY
                                              VALUE 10.
       05 SEQUENCE-LENGTH                    PICTURE 9(09) BINARY
                                              VALUE 0.
       05 SEQUENCE-BUFFER                    POINTER VALUE NULL.
       05 SEQUENCE-TYPE                      POINTER VALUE NULL.
*****
* Operation:      myop
* Mapped name:    myop
* Arguments:       <in> example/boundedseq myseq
* Returns:         example/boundedseq
* User Exceptions: none
*****
01 EXAMPLE-MYOP-ARGS.
   03 MYSEQ-1                                OCCURS 10 TIMES.
       05 MYSEQ                                PICTURE S9(10) BINARY.
   03 MYSEQ-SEQUENCE.
       05 SEQUENCE-MAXIMUM                    PICTURE 9(09) BINARY
                                              VALUE 10.
       05 SEQUENCE-LENGTH                    PICTURE 9(09) BINARY
```

**Example 14:** *COBOL Example for Bounded Sequences (Sheet 2 of 2)*

	VALUE 0.
05 SEQUENCE-BUFFER	POINTER VALUE NULL.
05 SEQUENCE-TYPE	POINTER VALUE NULL.
03 RESULT-1	OCCURS 10 TIMES.
05 RESULT	PICTURE S9(10) BINARY.
03 RESULT-SEQUENCE.	
05 SEQUENCE-MAXIMUM	PICTURE 9(09) BINARY
	VALUE 10.
05 SEQUENCE-LENGTH	PICTURE 9(09) BINARY
	VALUE 0.
05 SEQUENCE-BUFFER	POINTER VALUE NULL.
05 SEQUENCE-TYPE	POINTER VALUE NULL.

All elements of a bounded sequence can be accessed directly. Unpredictable results can occur if you access a sequence element that is past the current length but within the maximum number of elements for the sequence.

**IDL-to-COBOL mapping for unbounded sequences**

An unbounded sequence cannot map to a COBOL `OCCURS` clause, because the size of the sequence is not known. An incoming unbounded sequence is instead represented as a `USAGE IS POINTER` data item. A pointer is supplied that refers to an area of memory containing the sequence, which is not directly accessible. You must call `SEQGET` to copy a specified element of the sequence into an accessible data area. (Refer to [“SEQGET” on page 347](#) for more details). In this case, a group item is created to hold one element of the sequence, and a supporting group item is also created.

An outgoing 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 `SEQALLOC` function.

### Example of unbounded sequences mapping

The example can be broken down as follows:

1. Consider the following IDL:

```
// IDL
interface example
{
    typedef sequence<long> unboundedseq;
    attribute unboundedseq aseq;
    unboundedseq myop(in unboundedseq myseq);
};
```

2. The preceding IDL maps to the following COBOL:

#### Example 15: COBOL Example for Unbounded Sequences (Sheet 1 of 2)

```
*****
* Attribute:      aseq
* Mapped name:    aseq
* Type:           example/unboundedseq (read/write)
*****
01 EXAMPLE-ASEQ-ARGS.
   03 RESULT-1.
       05 RESULT                                PICTURE S9(10) BINARY.
   03 RESULT-SEQUENCE.
       05 SEQUENCE-MAXIMUM                      PICTURE 9(09) BINARY
                                                VALUE 0.
       05 SEQUENCE-LENGTH                      PICTURE 9(09) BINARY
                                                VALUE 0.
       05 SEQUENCE-BUFFER                      POINTER
                                                VALUE NULL.
       05 SEQUENCE-TYPE                        POINTER
                                                VALUE NULL.
*****
* Operation:      myop
* Mapped name:    myop
* Arguments:      <in> example/unboundedseq myseq
* Returns:        example/unboundedseq
* User Exceptions: none
*****
01 EXAMPLE-MYOP-ARGS.
   03 MYSEQ-1.
       05 MYSEQ                                PICTURE S9(10) BINARY.
   03 MYSEQ-SEQUENCE.
       05 SEQUENCE-MAXIMUM                      PICTURE 9(09) BINARY
                                                VALUE 0.
```

**Example 15:** *COBOL Example for Unbounded Sequences (Sheet 2 of 2)*

05 SEQUENCE-LENGTH	PICTURE 9(09) BINARY VALUE 0.
05 SEQUENCE-BUFFER	POINTER VALUE NULL.
05 SEQUENCE-TYPE	POINTER VALUE NULL.
03 RESULT-1.	
05 RESULT	PICTURE S9(10) BINARY.
03 RESULT-SEQUENCE.	
05 SEQUENCE-MAXIMUM	PICTURE 9(09) BINARY VALUE 0.
05 SEQUENCE-LENGTH	PICTURE 9(09) BINARY VALUE 0.
05 SEQUENCE-BUFFER	POINTER VALUE NULL.
05 SEQUENCE-TYPE	POINTER VALUE NULL.

Initial storage is assigned to the sequence via `SEQALLOC`. Elements of an unbounded sequence are not directly accessible. You can use `SEQGET` and `SEQSET` to access specific elements in the sequence.

**Note:** For details and examples of how to use the APIs pertaining to sequences, see [“SEQALLOC” on page 335](#), [“SEQDUP” on page 339](#), [“SEQFREE” on page 344](#), [“SEQGET” on page 347](#), and [“SEQSET” on page 350](#).

---

# Mapping for Array Type

---

## Overview

---

This section describes how arrays are mapped to COBOL.

---

## IDL-to-COBOL mapping for arrays

An IDL array definition maps directly to the COBOL `OCCURS` clause. Each element of the array is directly accessible.

---

## Example of IDL-to-COBOL

2. The preceding IDL maps to the following COBOL:

```
*****
* Attribute:      aarray
* Mapped name:    aarray
* Type:           example/long_array (read/write)
*****
01 EXAMPLE-AARRAY-ARGS.
   03 RESULT-1                                OCCURS 2 TIMES.
       05 RESULT-2                            OCCURS 5 TIMES.
           07 RESULT                          PICTURE S9(10) BINARY.
*****
* Operation:      myop
* Mapped name:    myop
* Arguments:      <in> example/long_array myarray
* Returns:        example/long_array
* User Exceptions: none
*****
01 EXAMPLE-MYOP-ARGS.
   03 MYARRAY-1                                OCCURS 2 TIMES.
       05 MYARRAY-2                            OCCURS 5 TIMES.
           07 MYARRAY                          PICTURE S9(10) BINARY.
   03 RESULT-1                                OCCURS 2 TIMES.
       05 RESULT-2                            OCCURS 5 TIMES.
           07 RESULT                          PICTURE S9(10) BINARY.
```

---

# Mapping for the Any Type

---

## Overview

This section describes how anys are mapped to COBOL.

---

## IDL-to-COBOL mapping for anys

The IDL `any` type maps to a COBOL pointer.

---

## Example of IDL-to-COBOL mapping for anys

The example can be broken down as follows:

1. Consider the following IDL:

```
// IDL
interface example
{
    typedef any a_any;
    attribute a_any aany;
    a_any myop(in a_any myany);
};
```

2. The preceding IDL maps to the following COBOL:

```
*****
* Attribute:      aany
* Mapped name:    aany
* Type:           example/a_any (read/write)
*****
01 EXAMPLE-AANY-ARGS.
   03 RESULT                                POINTER
                                           VALUE NULL.
*****
* Operation:      myop
* Mapped name:    myop
* Arguments:       <in> example/a_any myany
* Returns:         example/a_any
* User Exceptions: none
*****
01 EXAMPLE-MYOP-ARGS.
   03 MYANY                                POINTER
                                           VALUE NULL.

   03 RESULT                                POINTER
                                           VALUE NULL.
```

### Accessing and changing contents of an any

The contents of the `any` type cannot be accessed 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 the level 01 data name that is generated by the Orbix E2A 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`.

Refer to [“ANYGET” on page 272](#) and [“TYPEGET” on page 369](#) for details and an example of how to access the contents of an `any`. Refer to [“ANYSET” on page 274](#) and [“TYPESET” on page 371](#) for details and an example of how to change the contents of an `any`.



# Mapping for User Exception Type

## Overview

This section describes how user exceptions are mapped to COBOL.

## IDL-to-COBOL mapping for exceptions

An IDL exception maps to the following in COBOL:

- A level 01 group item that contains the definitions for all the user exceptions defined in the IDL. This group item is defined in COBOL as follows:

```
01 idlmembername-USER-EXCEPTIONS.
```

The group item contains the following level 03 items:

- ♦ An EXCEPTION-ID string that contains a textual description of the exception.
- ♦ A D data name that specifies the ordinal number of the current exception. Within this each user exception has a level 88 data name generated with its corresponding ordinal value.
- ♦ A U data name.
- ♦ A data name for each user exception, which redefines U. Within each of these data names are level 05 items that are the COBOL-equivalent user exception definitions for each user exception, based on the standard IDL-to-COBOL mapping rules.
- A level 01 data name with an EX-FQN-userexceptionname format, which has a string literal that uniquely identifies the user exception.
- A corresponding level 01 data name with an EX-FQN-userexceptionname-LENGTH format, which has a value specifying the length of the string literal.

**Note:** If D and U are used as IDL identifiers, they are treated as reserved words. This means that they are prefixed with IDL- in the generated COBOL. For example, the IDL identifier, d, maps to the COBOL identifier, IDL-D.

### Example of IDL-to-COBOL mapping for exceptions

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example {  
    exception bad {  
        long      value1;  
        string<32> reason;  
    };  
  
    exception worse {  
        short      value2;  
        string<16> errorcode;  
        string<32> reason;  
    };  
  
    void addName(in string name) raises(bad,worse);  
};
```

2. The preceding IDL maps to the following COBOL:

```
*****
* Operation:      AddName
* Mapped name:    AddName
* Arguments:      <in> string name
* Returns:        void
* User Exceptions: example/bad
*                  example/worse
*****
01 EXAMPLE-ADDNAME-ARGS.
   03 NAME                                     POINTER
                                           VALUE NULL.
*****
* User exception block
*****
01 EX-EXAMPLE-BAD                             PICTURE X(19)
                                           VALUE "IDL:example/bad:1.0".
01 EX-EXAMPLE-BAD-LENGTH                     PICTURE 9(09) BINARY
                                           VALUE 19.
01 EX-EXAMPLE-WORSE                          PICTURE X(21)
                                           VALUE "IDL:example/worse:1.0".
01 EX-EXAMPLE-WORSE-LENGTH                   PICTURE 9(09) BINARY
                                           VALUE 21.
01 EXAM16-USER-EXCEPTIONS.
   03 EXCEPTION-ID                           POINTER
                                           VALUE NULL.
   03 D                                       PICTURE 9(10) BINARY
                                           VALUE 0.
       88 D-NO-USEREXCEPTION                 VALUE 0.
       88 D-EXAMPLE-BAD                     VALUE 1.
       88 D-EXAMPLE-WORSE                   VALUE 2.
   03 U                                       PICTURE X(52)
                                           VALUE LOW-VALUES.
   03 EXCEPTION-EXAMPLE-BAD REDEFINES U.
       05 VALUE1                             PICTURE S9(10) BINARY.
       05 REASON                             PICTURE X(32).
   03 EXCEPTION-EXAMPLE-WORSE REDEFINES U.
       05 VALUE2                             PICTURE S9(05) BINARY.
       05 ERRORCODE                         PICTURE X(16).
       05 REASON                             PICTURE X(32).
```

## Raising a user exception

Use the `COAERR` function to raise a user exception. Refer to [“COAERR” on page 277](#) for more details.

---

# Mapping for Typedefs

## Overview

This section describes how typedefs are mapped to COBOL.

## IDL-to-COBOL mapping for typedefs

COBOL does not support typedefs directly. Any typedefs defined are output in the expanded form of the identifier that has been defined as a typedef, which is used in the group levels of the attributes and operations.

## Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example
{
    typedef fixed<8,2> millions;
    typedef struct database
    {
        string<40> full_name;
        long      date_of_birth;
        string<10> nationality;
        millions  income;
    } personnel;

    attribute millions dollars;
    personnel wages(in string employee_name, in millions
        new_salary);
};
```

2. Based on the preceding IDL, the attribute and operation argument buffer is generated as follows:

```
*****
* Attribute: dollars
* Mapped name: dollars
* Type: example/millions (read/write)
*****
01 EXAMPLE-DOLLARS-ARGS.
    03 RESULT PICTURE S9(6)V9(2) PACKED-DECIMAL.
*****
* Operation: wages
* Mapped name: wages
* Arguments: <in> string emp_name
* <in> example/millions new_salary
* Returns: example/personnel
* User Exceptions: none
*****
01 EXAMPLE-WAGES-ARGS.
    03 EMP-NAME POINTER                VALUE NULL.
    03 NEW-SALARY                      PICTURE S9(6)V9(2)
                                      PACKED-DECIMAL.

    03 RESULT.
        05 FULL-NAME                  PICTURE X(40).
        05 DATE-OF-BIRTH              PICTURE S9(10) BINARY.
        05 NATIONALITY                PICTURE X(10).
        05 INCOME                     PICTURE S9(6)V9(2)
                                      PACKED-DECIMAL.
```

3. Each typedef defined in the IDL is converted to a level 88 item in COBOL, in the typecode section. The string literal assigned to the level 88 item is the COBOL representation of the typecode for this type. These typecode key representations are used by COBOL applications when processing dynamic types such as sequences and anys.

```
*****
* Typecode section
* This contains CDR encodings of necessary typecodes.
*
*****
01 EXAM24-TYPE                                PICTURE X(25).
   COPY CORBATYP.
      88 EXAMPLE-PERSONNEL                    VALUE
         "IDL:example/personnel:1.0".
      88 EXAMPLE-MILLIONS                     VALUE
         "IDL:example/millions:1.0".
      88 EXAMPLE-DATABASE                     VALUE
         "IDL:example/database:1.0".
01 EXAM24-TYPE-LENGTH                        PICTURE S9(09) BINARY
                                              VALUE 25.
```

---

# Mapping for the Object Type

---

## Overview

This section describes how the `object` type is mapped to COBOL.

---

## IDL-to-COBOL mapping for typedefs

The IDL `object` type maps to a `POINTER` in COBOL.

---

## Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example
{
    typedef Object a_object;
    attribute a_object aobject;
    a_object myop(in a_object myobject);
};
```

2. The preceding IDL maps to the following COBOL:

```
*****
* Attribute:      aobject
* Mapped name:    aobject
* Type:           example/a_object (read/write)
*****
01 EXAMPLE-AOBJECT-ARGS.
   03 RESULT                                POINTER VALUE NULL.

*****
* Operation:      myop
* Mapped name:    myop
* Arguments:      <in> example/a_object myobject
* Returns:        example/a_object
* User Exceptions: none
*****
01 EXAMPLE-MYOP-ARGS.
   03 MY-OBJECT                                POINTER VALUE NULL.
   03 RESULT                                POINTER VALUE NULL.
```



# Mapping for Constant Types

## Overview

This section describes how constant types are mapped to COBOL.

## IDL-to-COBOL mapping for constants

Each set of `const` definitions at a different scope are given a unique 01 level COBOL name, where at root scope this name is `GLOBAL-idlmembername-CONSTS`. All other 01 levels are the fully scoped name of the module `/interface-CONSTS`.

You can use the `-o` argument with the Orbix E2A IDL compiler, to override the `idlmembername` with an alternative, user-defined name.

## Example

The example can be broken down as follows:

1. Consider the following IDL:

```
// IDL
const unsigned long myulong =1000;
const unsigned short myushort = 10;

module example
{
    const string<10> mystring="testing";

    interface example1
    {
        const long mylong =-1000;
        const short myshort = -10;
    };

    interface example2
    {
        const float myfloat =10.22;
        const double mydouble = 11.33;
    };
};
```

2. The preceding IDL maps to the following COBOL:

**Example 16:** *COBOL Example for Constant Types (Sheet 1 of 2)*

```
*****
* Constants in root scope:
*****
01 GLOBAL-EXAM18-CONSTS.
    03 MYULONG                                PICTURE 9(10) BINARY
                                              VALUE 1000.
    03 MYUSHORT                                PICTURE 9(05) BINARY
                                              VALUE 10.
*****
* Constants in example:
*****
01 EXAMPLE-CONSTS.
    03 MYSTRING                                PICTURE X(07)
                                              VALUE "testing".
*****
* Interface:
*   example/example1
*
* Mapped name:
*   example-example1
*
* Inherits interfaces:
*   (none)
*****
* Constants in example/example1:
*****
01 EXAMPLE-EXAMPLE1-CONSTS.
    03 MYLONG                                PICTURE S9(10) BINARY
                                              VALUE -1000.
    03 MYSHORT                                PICTURE S9(05) BINARY
                                              VALUE -10.
*****
* Interface:
*   example/example2
*
* Mapped name:
*   example-example2
*
* Inherits interfaces:
*   (none)
*****
```

**Example 16:** *COBOL Example for Constant Types (Sheet 2 of 2)*

```
*****
* Constants in example/example2:
*****
01 EXAMPLE-EXAMPLE2-CONSTS.
    03 MYFLOAT                      COMPUTATIONAL-1
                                      VALUE 1.022e+01.
    03 MYDOUBLE                     COMPUTATIONAL-2
                                      VALUE 1.133e+01.
```

# Mapping for Operations

## Overview

This section describes how IDL operations are mapped to COBOL.

## IDL-to-COBOL mapping for operations

An IDL operation maps to a number of statements in COBOL as follows:

1. A 01 group level is created for each operation. This group level is defined in the *idlmembername* copybook and contains a list of the parameters and the return type of the operation. If the parameters or the return type are of a dynamic type (for example, sequences, unbounded strings, or anys), no storage is assigned to them. The 01 group level is always suffixed by -ARGS (that is, *FQN-operationname-ARGS*).
2. A 01 level is created for each interface, with a PICTURE clause that contains the longest interface name of the interface operation(s) contained in the *idlmembername* copybook. The value of the PICTURE clause corresponds to the largest operation name length plus one, for example:

```
01 FQN-OPERATION          PICTURE X(maxoperationnamestring+1)
```

The extra space is added because the operation name must be terminated by a space when it is passed to the COBOL runtime by ORBEXEC.

A level 88 item is also created as follows for each operation, with a value clause that contains the string literal representing the operation name:

```
88 FQN-operationname      VALUE "operation-name-string".
```

A level 01 item is also created as follows, which defines the length of the maximum string representation of the interface operation:

```
01 FQN-OPERATION-LENGTH  PICTURE9(09) BINARY
                          VALUE maxoperationnamestring+1
```

3. The preceding identifiers in point 2 are referenced in a `select` clause that is generated in the *idlmembernameD* copybook. This `select` clause calls the appropriate operation paragraphs, which are discussed next.
4. The operation procedures are generated in the *idlmembernameS* source member when you specify the `-z` argument with the Orbix E2A IDL compiler. For example:
  - i. Consider the following IDL:

```
interface example
{
    long my_operation1(in long mylong);
    short my_operation2(in short myshort);
};
```

- ii. Based on the preceding IDL, the following COBOL is generated in the *idlmembername* copybook:

```
*****
* Operation: my_operation1
* Mapped name: my_operation1
* Arguments: <in> long mylong
* Returns: long
* User Exceptions: none
*****
01 EXAMPLE-MY-OPERATION1-ARGS.
   03 MYLONG PICTURE S9(10) BINARY.
   03 RESULT PICTURE S9(10) BINARY.
*****
* Operation: my_operation2
* Mapped name: my_operation2
* Arguments: <in> short myshort
* Returns: short
* User Exceptions: none
*****
01 EXAMPLE-MY-OPERATION2-ARGS.
   03 MYSHORT PICTURE S9(05) BINARY.
   03 RESULT PICTURE S9(05) BINARY.
```

- iii. The following code is also generated in the *idlmembername* copybook:

```
*****
*
* Operation List section
* This lists the operations and attributes which an
* interface supports
*
*****
01 EXAMPLE-OPERATION                PICTURE X(30).
   88 EXAMPLE-MY-OPERATION1         VALUE
      "my_operation1:IDL:example:1.0".
   88 EXAMPLE-MY-OPERATION2         VALUE
      "my_operation2:IDL:example:1.0".
01 EXAMPLE-OPERATION-LENGTH         PICTURE 9(09) BINARY
   VALUE 30.
```

- iv. The following code is generated in the *idlmembernameD* copybook member:

```
EVALUATE TRUE
  WHEN EXAMPLE-MY-OPERATION1
    PERFORM DO-EXAMPLE-MY-OPERATION1
  WHEN EXAMPLE-MY-OPERATION2
    PERFORM DO-EXAMPLE-MY-OPERATION2
END-EVALUATE
```

- v. The following is an example of the code in the *idlmembernameS* source member:

**Example 17: Server Mainline Example for Operations (Sheet 1 of 3)**

```
PROCEDURE DIVISION.
  ENTRY "DISPATCH".
  CALL "COAREQ" USING REQUEST-INFO.
  SET WS-COAREQ TO TRUE.
  PERFORM CHECK-STATUS.
  * Resolve the pointer reference to the interface name which
  * is the fully scoped interface name
  CALL "STRGET" USING INTERFACE-NAME
                      WS-INTERFACE-NAME-LENGTH
                      WS-INTERFACE-NAME.
  SET WS-STRGET TO TRUE.
  PERFORM CHECK-STATUS.
```

**Example 17: Server Mainline Example for Operations (Sheet 2 of 3)**

```

*****
* Interface(s) :
*****
    MOVE SPACES TO EXAMPLE-OPERATION.

*****
* Evaluate Interface(s) :
*****

    EVALUATE WS-INTERFACE-NAME
    WHEN 'IDL:example:1.0'

* Resolve the pointer reference to the operation information
    CALL "STRGET" USING OPERATION-NAME
                        EXAMPLE-OPERATION-LENGTH
                        EXAMPLE-OPERATION

    SET WS-STRGET TO TRUE
    PERFORM CHECK-STATUS
    END-EVALUATE.

COPY EXAM21D.
GOBACK.

DO-EXAMPLE-MY-OPERATION1.
    CALL "COAGET" USING EXAMPLE-MY-OPERATION1-ARGS.
    SET WS-COAGET TO TRUE.
    PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

    CALL "COAPUT" USING EXAMPLE-MY-OPERATION1-ARGS.
    SET WS-COAPUT TO TRUE.
    PERFORM CHECK-STATUS.

DO-EXAMPLE-MY-OPERATION2.
    CALL "COAGET" USING EXAMPLE-MY-OPERATION2-ARGS.
    SET WS-COAGET TO TRUE.
    PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

    CALL "COAPUT" USING EXAMPLE-MY-OPERATION2-ARGS.
    SET WS-COAPUT TO TRUE.
    PERFORM CHECK-STATUS.

```

**Example 17:** *Server Mainline Example for Operations (Sheet 3 of 3)*

```
*****  
* Check Errors Copybook  
*****  
COPY CHKERRS.
```



---

# Mapping for Attributes

---

## Overview

This section describes how IDL attributes are mapped to COBOL.

---

## IDL-to-COBOL mapping for attributes

IDL attributes are mapped to COBOL as level 88 items with a `_GET_` and `_SET_` prefix. Two level 88 items 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 level 88 item, with a `_GET_` prefix.

---

## Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example
{
    readonly attribute long mylong;
    attribute short myshort;
};
```

2. The preceding IDL maps to the following COBOL:

```
01 EXAMPLE-OPERATION                PICTURE X(29).
   88 EXAMPLE-GET-MYLONG             VALUE
      "_get_mylong:IDL:example:1.0".
   88 EXAMPLE-GET-MYSHORT            VALUE
      "_get_myshort:IDL:example:1.0".
   88 EXAMPLE-SET-MYSHORT            VALUE
      "_set_myshort:IDL:example:1.0".
01 EXAMPLE-OPERATION-LENGTH          PICTURE 9(09) BINARY
   VALUE 29.
```

# Mapping for Operations with a Void Return Type and No Parameters

**Overview** This section describes how IDL operations that have a void return type and no parameters are mapped to COBOL.

**Example** The example can be broken down as follows:

1. Consider the following IDL:

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

2. The preceding IDL maps to the following COBOL:

**Example 18:** *COBOL Example for Void Return Type (Sheet 1 of 2)*

```
*****
* Interface:
*   example
*
* Mapped name:
*   example
*
* Inherits interfaces:
*   (none)
*****
* Operation:      myoperation
* Mapped name:    myoperation
* Arguments:      None
* Returns:        void
* User Exceptions: none
*****
01 EXAMPLE-MYOPERATION-ARGS.
   03 FILLER                                PICTURE X(01).
*****
      COPY EXAM19X.
*****
```

**Example 18:** *COBOL Example for Void Return Type (Sheet 2 of 2)*

```

*****
*
* Operation List section
* This lists the operations and attributes which an
* interface supports
*
*****

01 EXAMPLE-OPERATION                                PICTURE X(28).
   88 EXAMPLE-MYOPERATION                            VALUE
      "myoperation:IDL:example:1.0".
01 EXAMPLE-OPERATION-LENGTH                          PICTURE 9(09)
                                                    BINARY VALUE 28.

```

**Note:** The filler is included for completeness, to allow the application to compile, but the filler is never actually referenced. The other code segments are generated as expected.

# Mapping for Inherited Interfaces

Overview	This section describes how inherited interfaces are mapped to COBOL.
IDL-to-COBOL 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 E2A IDL compiler generates an extra comment that contains a list of all the inherited interfaces.
Example	<p>The example can be broken down as follows:</p> <ol style="list-style-type: none"> <li>1. Consider the following IDL:</li> </ol>

```
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 maps to the following COBOL in the *idlmembername* copybook:

**Example 19:** *idlmembernameX Copybook Example (Sheet 1 of 4)*

```
*****
* Interface:
*   Account
*
* Mapped name:
*   Account
*
* Inherits interfaces:
*   (none)
*****
```

**Example 19:** *idlmembernameX Copybook Example (Sheet 2 of 4)*

```

*****
* Attribute:      mybaseshort
* Mapped name:    mybaseshort
* Type:           short (read/write)
*****
01 ACCOUNT-MYBASESHORT-ARGS.
    03 RESULT                                           PICTURE S9(05)
                                                         BINARY.
*****
* Operation:      mybasefunc
* Mapped name:     mybasefunc
* Arguments:       <in> long mybaselong
* Returns:         void
* User Exceptions: none
*****
01 ACCOUNT-MYBASEFUNC-ARGS.
    03 MYBASELONG                                       PICTURE S9(10)
                                                         BINARY.
*****
* Interface:
*   SavingAccount
*
* Mapped name:
*   SavingAccount
*
* Inherits interfaces:
*   Account
*****
* Attribute:      myshort
* Mapped name:     myshort
* Type:           short (read/write)
*****
01 SAVINGACCOUNT-MYSHORT-ARGS.
    03 RESULT                                           PICTURE S9(05)
                                                         BINARY.
*****
* Attribute:      mybaseshort
* Mapped name:     mybaseshort
* Type:           short (read/write)
*****
01 SAVINGACCOUNT-MYBASESHORT-ARGS.
    03 RESULT                                           PICTURE S9(05)
                                                         BINARY.

```

**Example 19:** *idlmembnameX Copybook Example (Sheet 3 of 4)*

```

*****
* Operation:      myfunc
* Mapped name:    myfunc
* Arguments:      <in> long mylong
* Returns:        void
* User Exceptions: none
*****
01 SAVINGACCOUNT-MYFUNC-ARGS.
    03 MYLONG                                           PICTURE S9(10)
                                                         BINARY.
*****
* Operation:      mybasefunc
* Mapped name:    mybasefunc
* Arguments:      <in> long mybaselong
* Returns:        void
* User Exceptions: none
*****
01 SAVINGACCOUNT-MYBASEFUNC-ARGS.
    03 MYBASELONG                                       PICTURE S9(10)
                                                         BINARY.
*****
*
* Operation List section
* This lists the operations and attributes which an
* interface supports
*
*****
01 ACCOUNT-OPERATION                                   PICTURE X(33).
    88 ACCOUNT-GET-MYBASESHORT                         VALUE
        "_get_mybaseshort:IDL:Account:1.0".
    88 ACCOUNT-SET-MYBASESHORT                         VALUE
        "_set_mybaseshort:IDL:Account:1.0".
    88 ACCOUNT-MYBASEFUNC                             VALUE
        "mybasefunc:IDL:Account:1.0".
01 ACCOUNT-OPERATION-LENGTH                           PICTURE 9(09)
                                                         BINARY VALUE 33.
01 SAVINGACCOUNT-OPERATION                            PICTURE X(39).
    88 SAVINGACCOUNT-GET-MYSHORT                       VALUE
        "_get_myshort:IDL:SavingAccount:1.0".
    88 SAVINGACCOUNT-SET-MYSHORT                       VALUE
        "_set_myshort:IDL:SavingAccount:1.0".
    88 SAVINGACCOUNT-MYFUNC                           VALUE
        "myfunc:IDL:SavingAccount:1.0".
    88 SAVINGACCOUNT-GET-MYBASESHORT                   VALUE
        "_get_mybaseshort:IDL:SavingAccount:1.0".

```

**Example 19:** *idlmembernameX Copybook Example (Sheet 4 of 4)*

```

88 SAVINGACCOUNT-SET-MYBASESHORT      VALUE
   "_set_mybaseshort:IDL:SavingAccount:1.0".
88 SAVINGACCOUNT-MYBASEFUNC           VALUE
   "mybasefunc:IDL:SavingAccount:1.0".
01 SAVINGACCOUNT-OPERATION-LENGTH     PICTURE 9(09)
                                       BINARY VALUE 39.

```

3. The following code is generated in the *idlmembernameD* copybook:

```

EVALUATE TRUE
  WHEN ACCOUNT-GET-MYBASESHORT
    PERFORM DO-ACCOUNT-GET-MYBASESHORT
  WHEN ACCOUNT-SET-MYBASESHORT
    PERFORM DO-ACCOUNT-SET-MYBASESHORT
  WHEN ACCOUNT-MYBASEFUNC
    PERFORM DO-ACCOUNT-MYBASEFUNC
  WHEN SAVINGACCOUNT-GET-MYSHORT
    PERFORM DO-SAVINGACCOUNT-GET-MYSHORT
  WHEN SAVINGACCOUNT-SET-MYSHORT
    PERFORM DO-SAVINGACCOUNT-SET-MYSHORT
  WHEN SAVINGACCOUNT-MYFUNC
    PERFORM DO-SAVINGACCOUNT-MYFUNC
  WHEN SAVINGACCOUNT-GET-MYBASESHORT
    PERFORM DO-SAVINGACCOUNT-GET-MYBA-6FF2
  WHEN SAVINGACCOUNT-SET-MYBASESHORT
    PERFORM DO-SAVINGACCOUNT-SET-MYBA-AE11
  WHEN SAVINGACCOUNT-MYBASEFUNC
    PERFORM DO-SAVINGACCOUNT-MYBASEFUNC
END-EVALUATE

```

4. The following is an example of the code in the *idlmembernameS* server implementation program:

**Example 20: Server Mainline Example (Sheet 1 of 3)**

```
*****
* Interface(s) :
*****
      MOVE SPACES TO ACCOUNT-OPERATION.
      MOVE SPACES TO SAVINGACCOUNT-OPERATION.

*****
* Evaluate Interface(s) :
*****

      EVALUATE WS-INTERFACE-NAME
      WHEN 'IDL:Account:1.0'

* Resolve the pointer reference to the operation information
      CALL "STRGET" USING OPERATION-NAME
                           ACCOUNT-OPERATION-LENGTH
                           ACCOUNT-OPERATION
      SET WS-STRGET TO TRUE
      PERFORM CHECK-STATUS
      WHEN 'IDL:SavingAccount:1.0'

* Resolve the pointer reference to the operation information
      CALL "STRGET" USING OPERATION-NAME
                           SAVINGACCOUNT-OPERATION-LENGTH
                           SAVINGACCOUNT-OPERATION
      SET WS-STRGET TO TRUE
      PERFORM CHECK-STATUS
      END-EVALUATE.

COPY EXAM20D.
GOBACK.

DO-ACCOUNT-GET-MYBASESHORT.
      CALL "COAGET" USING ACCOUNT-MYBASESHORT-ARGS.
      SET WS-COAGET TO TRUE.
      PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

      CALL "COAPUT" USING ACCOUNT-MYBASESHORT-ARGS.
      SET WS-COAPUT TO TRUE.
      PERFORM CHECK-STATUS.
```



**Example 20: Server Mainline Example (Sheet 2 of 3)**

```

DO-ACCOUNT-SET-MYBASESHORT.
    CALL "COAGET" USING ACCOUNT-MYBASESHORT-ARGS.
    SET WS-COAGET TO TRUE.
    PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

    CALL "COAPUT" USING ACCOUNT-MYBASESHORT-ARGS.
    SET WS-COAPUT TO TRUE.
    PERFORM CHECK-STATUS.
DO-ACCOUNT-MYBASEFUNC.
    CALL "COAGET" USING ACCOUNT-MYBASEFUNC-ARGS.
    SET WS-COAGET TO TRUE.
    PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

    CALL "COAPUT" USING ACCOUNT-MYBASEFUNC-ARGS.
    SET WS-COAPUT TO TRUE.
    PERFORM CHECK-STATUS.
DO-SAVINGACCOUNT-GET-MYSHORT.
    CALL "COAGET" USING SAVINGACCOUNT-MYSHORT-ARGS.
    SET WS-COAGET TO TRUE.
    PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

    CALL "COAPUT" USING SAVINGACCOUNT-MYSHORT-ARGS.
    SET WS-COAPUT TO TRUE.
    PERFORM CHECK-STATUS.
DO-SAVINGACCOUNT-SET-MYSHORT.
    CALL "COAGET" USING SAVINGACCOUNT-MYSHORT-ARGS.
    SET WS-COAGET TO TRUE.
    PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

    CALL "COAPUT" USING SAVINGACCOUNT-MYSHORT-ARGS.
    SET WS-COAPUT TO TRUE.
    PERFORM CHECK-STATUS.
DO-SAVINGACCOUNT-MYFUNC.
    CALL "COAGET" USING SAVINGACCOUNT-MYFUNC-ARGS.
    SET WS-COAGET TO TRUE.
    PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

```

**Example 20: Server Mainline Example (Sheet 3 of 3)**

```

CALL "COAPUT" USING SAVINGACCOUNT-MYFUNC-ARGS.
SET WS-COAPUT TO TRUE.
PERFORM CHECK-STATUS.
DO-SAVINGACCOUNT-GET-MYBA-6FF2.
CALL "COAGET" USING SAVINGACCOUNT-MYBASESHORT-ARGS.
SET WS-COAGET TO TRUE.
PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

CALL "COAPUT" USING SAVINGACCOUNT-MYBASESHORT-ARGS.
SET WS-COAPUT TO TRUE.
PERFORM CHECK-STATUS.
DO-SAVINGACCOUNT-SET-MYBA-AE11.
CALL "COAGET" USING SAVINGACCOUNT-MYBASESHORT-ARGS.
SET WS-COAGET TO TRUE.
PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

CALL "COAPUT" USING SAVINGACCOUNT-MYBASESHORT-ARGS.
SET WS-COAPUT TO TRUE.
PERFORM CHECK-STATUS.
DO-SAVINGACCOUNT-MYBASEFUNC.
CALL "COAGET" USING SAVINGACCOUNT-MYBASEFUNC-ARGS.
SET WS-COAGET TO TRUE.
PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

CALL "COAPUT" USING SAVINGACCOUNT-MYBASEFUNC-ARGS.
SET WS-COAPUT TO TRUE.
PERFORM CHECK-STATUS.

*****
* Check Errors Copybook
*****
COPY CHKERRS.

```

# Mapping for Multiple Interfaces

## Overview

This section describes how multiple interfaces are mapped to COBOL.

## Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example1
{
    readonly attribute long mylong;
    attribute short myshort;
};

interface example2
{
    readonly attribute long mylong;
    attribute short myshort;
};
```

2. Based on the preceding IDL, the following code is generated in the `idlmembernameS` member:

### Example 21: Server Implementation Example (Sheet 1 of 3)

```
ENTRY "DISPATCH".
CALL "COAREQ" USING REQUEST-INFO.
SET WS-COAREQ TO TRUE.
PERFORM CHECK-STATUS.
* Resolve the pointer reference to the interface name which
* is the fully scoped interface name
CALL "STRGET" USING INTERFACE-NAME
                        WS-INTERFACE-NAME-LENGTH
                        WS-INTERFACE-NAME.
SET WS-STRGET TO TRUE.
PERFORM CHECK-STATUS.

*****
* Interface(s) :
*****
MOVE SPACES TO EXAMPLE1-OPERATION.
MOVE SPACES TO EXAMPLE2-OPERATION.
```

**Example 21: Server Implementation Example (Sheet 2 of 3)**

```

*****
* Evaluate Interface(s) :
*****

      EVALUATE WS-INTERFACE-NAME
      WHEN 'IDL:example1:1.0'

* Resolve the pointer reference to the operation information
      CALL "STRGET" USING OPERATION-NAME
                           EXAMPLE1-OPERATION-LENGTH
                           EXAMPLE1-OPERATION

      SET WS-STRGET TO TRUE
      PERFORM CHECK-STATUS
      WHEN 'IDL:example2:1.0'

* Resolve the pointer reference to the operation information
      CALL "STRGET" USING OPERATION-NAME
                           EXAMPLE2-OPERATION-LENGTH
                           EXAMPLE2-OPERATION

      SET WS-STRGET TO TRUE
      PERFORM CHECK-STATUS
      END-EVALUATE.

COPY EXAM23D.
GOBACK.

DO-EXAMPLE1-GET-MYLONG.
      CALL "COAGET" USING EXAMPLE1-MYLONG-ARGS.
      SET WS-COAGET TO TRUE.
      PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

      CALL "COAPUT" USING EXAMPLE1-MYLONG-ARGS.
      SET WS-COAPUT TO TRUE.
      PERFORM CHECK-STATUS.
DO-EXAMPLE1-GET-MYSHORT.
      CALL "COAGET" USING EXAMPLE1-MYSHORT-ARGS.
      SET WS-COAGET TO TRUE.
      PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

      CALL "COAPUT" USING EXAMPLE1-MYSHORT-ARGS.
      SET WS-COAPUT TO TRUE.

```

**Example 21: Server Implementation Example (Sheet 3 of 3)**

```

PERFORM CHECK-STATUS.
DO-EXAMPLE1-SET-MYSHORT.
  CALL "COAGET" USING EXAMPLE1-MYSHORT-ARGS.
  SET WS-COAGET TO TRUE.
  PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

  CALL "COAPUT" USING EXAMPLE1-MYSHORT-ARGS.
  SET WS-COAPUT TO TRUE.
  PERFORM CHECK-STATUS.
DO-EXAMPLE2-GET-MYLONG.
  CALL "COAGET" USING EXAMPLE2-MYLONG-ARGS.
  SET WS-COAGET TO TRUE.
  PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

  CALL "COAPUT" USING EXAMPLE2-MYLONG-ARGS.
  SET WS-COAPUT TO TRUE.
  PERFORM CHECK-STATUS.
DO-EXAMPLE2-GET-MYSHORT.
  CALL "COAGET" USING EXAMPLE2-MYSHORT-ARGS.
  SET WS-COAGET TO TRUE.
  PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

  CALL "COAPUT" USING EXAMPLE2-MYSHORT-ARGS.
  SET WS-COAPUT TO TRUE.
  PERFORM CHECK-STATUS.
DO-EXAMPLE2-SET-MYSHORT.
  CALL "COAGET" USING EXAMPLE2-MYSHORT-ARGS.
  SET WS-COAGET TO TRUE.
  PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

  CALL "COAPUT" USING EXAMPLE2-MYSHORT-ARGS.
  SET WS-COAPUT TO TRUE.
  PERFORM CHECK-STATUS.
*****
* Check Errors Copybook
*****
COPY CHKERRS.

```



# Orbix E2A IDL Compiler

*This chapter describes the Orbix E2A IDL compiler in terms of the JCL used to run it, the COBOL members that it creates, the arguments that you can use with it, and the configuration settings that it uses.*

## In this chapter

This chapter discusses the following topics:

<a href="#">Running the Orbix E2A IDL Compiler</a>	<a href="#">page 238</a>
<a href="#">Generated COBOL Members</a>	<a href="#">page 240</a>
<a href="#">Orbix E2A IDL Compiler Arguments</a>	<a href="#">page 242</a>
<a href="#">Configuration Member Settings</a>	<a href="#">page 257</a>

**Note:** The supplied demonstrations include examples of JCL that can be used to run the Orbix E2A 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 your installation's high-level qualifier.

# Running the Orbix E2A IDL Compiler

## The Orbix E2A IDL compiler

You can use the Orbix E2A IDL compiler to generate COBOL source and copybooks from IDL definitions.

## Orbix E2A IDL compiler configuration

The Orbix E2A IDL compiler uses the Orbix E2A configuration member for its settings. The JCL that runs the compiler uses the `IDL` member in the `orbixhlq.CONFIG` configuration PDS.

## Running the Orbix E2A IDL compiler

For the purposes of this example, the COBOL source is generated in the first step of the following job (that is, the JCL supplied with the `simple` server demonstration):

```
orbixhlq.DEMOS.COBOL.BUILD.JCL(SIMPLIDL)
```

## Example of the SIMPLIDL JCL

The following is the supplied JCL to run the Orbix E2A IDL compiler for the `simple` demonstration:

```
//SIMPLIDL JOB  (),
//          CLASS=A,
//          MSGCLASS=X,
//          MSGLEVEL=(1,1),
//          REGION=0M,
//          TIME=1440,
//          NOTIFY=&SYSUID,
//          COND=(4,LT)
//*-----
/* Orbix E2A - Generate the COBOL copybooks for the Simple
/* Client
/*-----
//          JCLLIB ORDER=(orbixhlq..PROCS)
//          INCLUDE MEMBER=(ORXVARS)
/*
//IDLCBL   EXEC ORXIDL,
//          SOURCE=SIMPLE,
//          IDL=&ORBIX..DEMOS.IDL,
//          COPYLIB=&ORBIX..DEMOS.COBOL.COPYLIB,
//          IMPL=&ORBIX..DEMOS.COBOL.SRC,
//          IDLPARM=' -cobol'
```



The preceding JCL generates COBOL copybooks from an IDL member called `SIMPLE` (see the `SOURCE=SIMPLE` line). This JCL does not specify any compiler arguments (see the `IDLPARM` line); therefore, it cannot generate any COBOL source code members, which can only be generated if you specify the `-s` and `-z` arguments. See [“Specifying the Compiler Arguments” on page 243](#) for details of how to do this.

**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 the Orbix E2A installation, which is set in <code>orbixh1q.PROCS(ORXVARS)</code> .
SOURCE	The IDL member to be compiled.
IDL	The PDS for the IDL member.
COPYLIB	The PDS for the COBOL copybooks generated by the Orbix E2A IDL compiler.
IMPL	The PDS for the COBOL source code members generated by the Orbix E2A IDL compiler.
IDLPARM	The plug-in to the Orbix E2A IDL compiler to be used (in the preceding example, it is the COBOL plug-in), and any arguments to be passed to it (in the preceding example, no arguments are specified). See <a href="#">“Specifying the Compiler Arguments” on page 243</a> for details of how to specify the Orbix E2A IDL compiler arguments as parameters to it.

# Generated COBOL Members

**Overview** This section describes the various COBOL source code and copybook members that the Orbix E2A IDL compiler generates.

**Generated members** [Table 19](#) provides an overview and description of the COBOL members that the Orbix E2A IDL compiler generates, based on the IDL member name.

**Table 19:** *COBOL Members Generated by the Orbix E2A IDL Compiler*

Member Name	Member Type	Compiler Argument Used to Generate	Description
<i>idlmembernameS</i>	Source code	-z	This is server implementation source code member. It contains stub paragraphs for all the callable operations. It is only generated if you specify the -z argument.
<i>idlmembernameSV</i>	Source code	-s	This is the server mainline source code member. It is only generated if you specify the -s argument.
<i>idlmembername</i>	Copybook	Generated by default	This copybook contains data definitions that are used for working with operation parameters and return values for each interface defined in the IDL member.
<i>idlmembernameX</i>	Copybook	Generated by default	This copybook contains data definitions that are used by the Orbix COBOL runtime to support the interfaces defined in the IDL member. This copybook is automatically included in the <i>idlmembername</i> copybook.
<i>idlmembernameD</i>	Copybook	Generated by default	This copybook contains procedural code for performing the correct paragraph for the request operation. This copybook is automatically included in the <i>idlmembernameS</i> source code member.

---

**Member name restrictions**

If the IDL member name exceeds six characters, the Orbix E2A IDL compiler uses only the first six characters of the IDL member name when generating the source and copybook member names. This allows space for appending the two-character *sv* suffix to the name for the server mainline code member, while allowing it to adhere to the eight-character maximum size limit for OS/390 member names. In such cases, each of the other generated member names is also based on only the first six characters of the IDL member name, and is appended with its own suffix, as appropriate.

---

# Orbix E2A IDL Compiler Arguments

---

**Overview** This section describes the various arguments that you can specify as parameters to the Orbix E2A IDL compiler.

---

**Summary of the arguments** The Orbix E2A IDL compiler arguments can be summarized as follows:

- Q     Indicate whether single or double quotes are to be used for string literals in COBOL copybooks.
- M     Set up an alternative mapping scheme for data names.
- Z     Generate server implementation source code.
- S     Generate server mainline source code.
- T     Indicate whether server code is for batch, IMS, or CICS.
- O     Override default copybook names with a different name.

All these arguments are optional. This means that they do not have to be specified as parameters to the compiler.

---

**In this section** This section discusses the following topics:

<a href="#">Specifying the Compiler Arguments</a>	<a href="#">page 243</a>
<a href="#">-M Argument</a>	<a href="#">page 244</a>
<a href="#">-O Argument</a>	<a href="#">page 250</a>
<a href="#">-Q Argument</a>	<a href="#">page 252</a>
<a href="#">-S Argument</a>	<a href="#">page 253</a>
<a href="#">-T Argument</a>	<a href="#">page 254</a>
<a href="#">-Z Argument</a>	<a href="#">page 256</a>

---

## Specifying the Compiler Arguments

---

### Overview

This section describes how to specify the available arguments as parameters to the Orbix E2A IDL compiler.

---

### The IDLPARM DD name

To denote the arguments that you want to specify as parameters to the compiler, you can use the DD name, `IDLPARM`, in the JCL that you use to run the compiler. See [“Running the Orbix E2A IDL Compiler” on page 238](#) for an example of the JCL to run the Orbix E2A IDL compiler for the supplied `SIMPLE` IDL member.

---

### IDLPARM line format

The parameters for the `IDLPARM` entry in the JCL take the following format:

```
// IDLPARM='-cobol[:-M[option][membername]][:-Omembername]
[: -Q[option]][: -S][: -T[option]][: -Z]'
```

Each argument must be preceded by a colon followed by a hyphen (that is, `:-`), with no spaces between any characters or any arguments.

**Note:** If you set `IsDefault=YES` in the COBOL section of the `orbixhlq.CONFIG(IDL)` configuration member, you do not need to specify the `-cobol` switch in the `IDLPARM` line of the JCL. See [“Configuration Member Settings” on page 257](#) for more details.

## -M Argument

### Description

COBOL data names generated by the Orbix E2A IDL compiler are based on fully qualified interface names by default (that is, *IDLmodule(s)-IDLinterfacename-IDLvariablename*). You can use the `-M` argument with the Orbix E2A IDL compiler to define your own alternative mapping scheme for data names. This is particularly useful if your COBOL data names are likely to exceed the 30-character restriction imposed by the COBOL compiler.

### IDLMAP DD card

To use the `-M` argument, you must define a DD card for `IDLMAP` in the JCL that you use to run the Orbix E2A IDL compiler. This DD card specifies the PDS for the mapping member generated by the Orbix E2A IDL compiler. For example, you might define the DD card as follows in the JCL:

```
//IDLMAP DD DISP=SHR,DSN=HLQ.ASP50.DEMOS.COBOL.MAP
```

You can define a DD card for `IDLMAP` even if you do not specify the `-M` argument as a parameter to the Orbix E2A IDL compiler. The DD card is simply ignored if the `-M` argument is not specified.

### Example of data names generated by default

The example can be broken down as follows:

1. Consider the following IDL:

```
module Banks{
    module IrishBanks{
        interface SavingsBank{attribute short accountbal;};
        interface NationalBank{};
        interface DepositBank{};
    };
};
```

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates by default the data names shown in [Table 20](#) for the specified interfaces:

**Table 20:** *Example of Default Generated Data Names*

Interface Name	Generated Data Name
SavingsBank	Banks-IrishBanks-SavingsBank
NationalBank	Banks-IrishBanks-NationalBank
DepositBank	Banks-IrishBanks-DepositBank

By using the `-M` argument, you can replace the fully scoped names shown in [Table 20](#) with alternative data names.

**Steps to generate alternative names with the `-M` argument**

The steps to generate alternative data name mappings with the `-M` argument are:

Step	Action
1	Run the Orbix E2A IDL compiler with the <code>-Mcreate</code> argument, to generate the mapping member, complete with the fully qualified names and their alternative mappings.
2	Edit (if necessary) the generated mapping member, to change the alternative name mappings to the names you want to use.
3	Run the Orbix E2A IDL compiler with the <code>-Mprocess</code> argument, to generate COBOL copybooks with the alternative data names.

**Step 1—Generate the mapping member**

First, you must run the Orbix E2A IDL compiler with the `-Mcreate` argument, to generate the mapping member, which contains the fully qualified names and the alternative name mappings. The format of the command in the IDL compiler JCL is as follows (where *x* represents the scope level, and *BANK* is the name of the mapping member you want to create):

```
IDLPARM= '-cobol:-McreateXBANK',
```

**Explanation of the command in step 1**

The components of the preceding command can be explained as follows (note that there must be no spaces between these components):

- 
- `create` This specifies that the `-M` argument is to create a mapping member.
- `x` 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 COBOL names (that is, to the IDL variable name only).
  - 1 Map fully scoped IDL names to partially scoped COBOL names (that is, to *IDLinterfacename-IDLvariablename*). The scope operator, `/`, is replaced with a hyphen, `-`.
  - 2 Map fully scoped IDL names to fully scoped COBOL names (that is, to *IDLmodule(s)-IDLinterfacename-IDLvariablename*). The scope operator, `/`, is replaced with a hyphen, `-`.
- `BANK` This is the name of the mapping member to be created. It 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.
- 

**Examples of scoping levels described in step 1**

The level of scoping within the generated alternative name mappings is determined by whether you specify 0, 1, or 2 with the `-Mcreate` command.

The example can be broken down as follows:

1. Consider the following IDL:

```
module Banks{
    module IrishBanks{
        interface SavingsBank{attribute short accountbal;};
        interface NationalBank{void deposit
            (in long amount);};
    };
};
```



2. Based on the preceding IDL example, a `-Mcreate0BANK` command produces the `BANK` mapping member contents shown in [Table 21](#).

**Table 21:** *Example of Level-0-Scoped Generated Data Names*

Fully Scoped IDL Names	Generated Alternative Names
Banks	Banks
Banks/IrishBanks	IrishBanks
Banks/IrishBanks/SavingsBank	SavingsBank
Banks/IrishBanks/SavingsBank/ accountbal	accountbal
Banks/IrishBanks/NationalBank	NationalBank
Banks/IrishBanks/NationalBank/ deposit	deposit

Alternatively, based on the preceding IDL example, a `-Mcreate1BANK` command produces the `BANK` mapping member contents shown in [Table 22](#).

**Table 22:** *Example of Level-1-Scoped Generated Data Names*

Fully Scoped IDL Names	Generated Alternative Names
Banks	Banks
Banks/IrishBanks	IrishBanks
Banks/IrishBanks/SavingsBank	SavingsBank
Banks/IrishBanks/SavingsBank/ accountbal	SavingsBanks-accountbal
Banks/IrishBanks/NationalBank	NationalBank
Banks/IrishBanks/NationalBank/ deposit	NationalBank-deposit

Alternatively, based on the preceding IDL example, a `-Mcreate2BANK` command produces the `BANK` mapping member contents shown in [Table 23](#).

**Table 23:** *Example of Level-2-Scoped Generated Data Names*

Fully Scoped IDL Names	Generated Alternative Names
Banks	Banks
Banks/IrishBanks	Banks-IrishBanks
Banks/IrishBanks/SavingsBank	Banks-IrishBanks-SavingsBank
Banks/IrishBanks/SavingsBank/ accountbal	Banks-IrishBanks-SavingsBanks- accountbal
Banks/IrishBanks/NationalBank	Banks-IrishBanks-NationalBank
Banks/IrishBanks/NationalBank/ deposit	Banks-IrishBanks-NationalBank- deposit

**Step 2—Change the alternative name mappings**

You can manually edit the mapping member to change the alternative names to the names that you want to use. For example, you might change the mappings in the `BANK` mapping member as follows:

Fully Scoped IDL Names	Modified Names
Banks/IrishBanks	IrishBanks
Banks/IrishBanks/SavingsBank	MyBank
Banks/IrishBanks/NationalBank	MyOtherBank
Banks/IrishBanks/SavingsBank/accountbal	Myaccountbalance

Note the following rules:

- The fully scoped name and the alternative name meant to replace it must be separated by one space (and one space only).
- 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.
- If the alternative name exceeds 30 characters, it is abbreviated to 30 characters, subject to the normal COBOL mapping rules for identifiers.

### Step 3—Generate the COBOL copybooks

When you have changed the alternative mapping names as necessary, run the Orbix E2A IDL compiler with the `-Mprocess` argument, to generate your COBOL copybooks complete with the alternative data names that you have set up in the specified mapping member.

The following command generates data names for the contents of the specified IDL member, based on the alternative name mappings in the `BANK` mapping member:

```
IDLPARM= '-cobol:-MprocessBANK'
```

When you run the preceding command, your COBOL copybooks are generated with the alternative data names you want to use, instead of with the fully qualified data names that the Orbix E2A IDL compiler generates by default.

---

## -O Argument

### Description

COBOL source and copybook names generated by the Orbix E2A IDL compiler are based by default on the IDL member name. You can use the `-o` argument with the Orbix E2A IDL compiler to map the default generated member names to an alternative naming scheme, if you wish.

The `-o` argument is, for example, particularly useful for users who have migrated from IONA's Orbix 2.3-based solution for OS/390, and who want to avoid having to change the `COPY` statements in their existing application source code. In this case, they can use the `-o` argument to automatically change the generated member names to the alternative names they want to use.

**Note:** If you are an existing user who has migrated from IONA's Orbix 2.3-based solution for OS/390, see the *Mainframe Migration Guide* for more details.

### Example of copybooks generated by Orbix E2A IDL compiler

The example can be broken down as follows:

1. Consider the following IDL contained in the `TEST` IDL member:

```
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 E2A IDL compiler generates the following COBOL copybooks, based on the IDL member name:
  - ♦ `TEST`
  - ♦ `TESTX`
  - ♦ `TESTD`

## Specifying the -O argument

---

The following piece of JCL changes the copybook names from `TEST`

---

---

# -Q Argument

Description	The -Q argument indicates whether single or double quotes are to be used on string literals in COBOL copybooks.
Qualifying parameters	The -Q argument must be qualified by either s or d. If you specify -Qs, single quotes are used. If you specify -Qd, double quotes are used. If you do not specify the -Q argument, double quotes are used by default.
Specifying the -Q argument	<div><p>The following piece of JCL specifies that single quotes are to be used on string literals in COBOL copybooks that are generated from the SIMPLE IDL member:</p><pre>// SOURCE=SIMPLE , // ... // IDLPARM=' -cobol : -Qs '</pre></div>

---

## -S Argument

### Description

The `-s` argument generates the server mainline source code member (that is, the `idlmembnameSV` program). This program is not generated by default by the Orbix E2A IDL compiler. It is only generated if you use the `-s` argument, because doing so overwrites any server code that has already been created based on that IDL member name.

**WARNING:** Only specify the `-s` argument if you want to generate a new server mainline source code member or overwrite an existing one.

### Specifying the `-S` argument

The following piece of JCL creates a server mainline program called `SIMPLESV`, based on the `SIMPLE` IDL member:

```
// SOURCE=SIMPLE
// ...
// IDLPARM=' -cobol:-S'.
```

---

# -T Argument

Description	<div data-bbox="475 338 1250 442"><p>The <code>-T</code> argument allows you to specify whether the server code you want to generate is for use in batch, IMS, or CICS. The valid options for this argument are:</p></div> <div data-bbox="475 442 1250 612"><table><tr><td data-bbox="475 442 602 477">NATIVE</td><td data-bbox="602 442 1250 612"><p>Specifying <code>-TNATIVE</code> with <code>-S</code> generates batch server mainline code. Specifying <code>-TNATIVE</code> with <code>-Z</code> generates batch server implementation code.</p><p>Specifying <code>-TNATIVE</code> is the same as not specifying <code>-T</code> at all. That is, unless you specify <code>-TIMS</code>,</p></td></tr></table></div>	NATIVE	<p>Specifying <code>-TNATIVE</code> with <code>-S</code> generates batch server mainline code. Specifying <code>-TNATIVE</code> with <code>-Z</code> generates batch server implementation code.</p> <p>Specifying <code>-TNATIVE</code> is the same as not specifying <code>-T</code> at all. That is, unless you specify <code>-TIMS</code>,</p>
NATIVE	<p>Specifying <code>-TNATIVE</code> with <code>-S</code> generates batch server mainline code. Specifying <code>-TNATIVE</code> with <code>-Z</code> generates batch server implementation code.</p> <p>Specifying <code>-TNATIVE</code> is the same as not specifying <code>-T</code> at all. That is, unless you specify <code>-TIMS</code>,</p>		

---

Specifying the <code>-T</code> argument for batch	<p>The following piece of JCL creates a batch COBOL server mainline program (called <code>SIMPLESV</code>) and a batch COBOL server implementation program (called <code>SIMPLES</code>), based on the <code>SIMPLE</code> IDL member:</p>
---	--



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

See [“Developing the Server” on page 24](#) for an example of batch COBOL server mainline and implementation programs.

---

### Specifying the `-T` argument for IMS

The following piece of JCL creates an IMS COBOL server mainline program (called `SIMPLESV`) and an IMS COBOL server implementation program (called `SIMPLES`), based on the `SIMPLE` IDL member:

```
// SOURCE=SIMPLE ,  
// ...  
// IDLPARM='-cobol:-S:-Z:-TIMS' ,
```

See [“Developing the Server” on page 62](#) for an example of IMS COBOL server mainline and implementation programs.

---

### Specifying the `-T` argument for CICS

The following piece of JCL creates a CICS COBOL server mainline program (called `SIMPLESV`) and a CICS COBOL server implementation program (called `SIMPLES`), based on the `SIMPLE` IDL member:

```
// SOURCE=SIMPLE ,  
// ...  
// IDLPARM='-cobol:-S:-Z:-TCICS' ,
```

See [“Developing the Server” on page 95](#) for an example of CICS COBOL server mainline and implementation programs.

---

## -Z Argument

### Description

---

The `-z` argument generates the skeleton server implementation source code member (that is, the `idlmembernameS` program). The generated code contains stub paragraphs for all the callable operations in the defined IDL. This program is not generated by default. It is only generated if you use the `-z` argument, because doing so overwrites any server implementation code that has already been createh

---

---

# Configuration Member Settings

---

## Overview

This section describes the configuration settings for the Orbix E2A IDL compiler `-cobo1` plug-in for COBOL source code and copybook generation, and the `-mfa` plug-in for IMS 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:

<a href="#">COBOL Configuration Settings</a>	<a href="#">page 258</a>
<a href="#">Adapter Mapping Member Configuration Settings</a>	<a href="#">page 261</a>

---

## COBOL Configuration Settings

---

### Overview

The `orbixhlg.CONFIG(IDL)` member contains settings for COBOL, along with those for C++ and several other languages.

---

### Configuration settings

The COBOL configuration is listed under `Cobol` as follows:

```
Cobol
{
    Switch = "cobol";
    ShlibName = "ORXBCBL";
    ShlibMajorVersion = "x";
    IsDefault = "NO";
    PresetOptions = "";

#    COBOL files and copybooks extensions
#    The Default is .cbl, .xxx and .cpy on NT and none for
#    OS/390.
    CobolExtension = "";
    ImplementationExtension = "";
    CopybookExtension = "";
};
```

**Note:** Settings listed with a # are considered to be comments and are not in effect.

---

### Mandatory settings

The first three of the preceding settings are mandatory and must not be altered. They inform the Orbix E2A IDL compiler how to recognize the COBOL 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 settings

Table 24 provides an overview and description of the available configuration settings.

**Table 24:** *COBOL Configuration Settings (Sheet 1 of 2)*

Setting Name	Description	Default
IsDefault	Indicates whether COBOL is the language that the Orbix E2A IDL compiler generates by default from IDL. If this is set to YES, you do not need to specify the <code>-cobol</code> switch when running the compiler.	
PresetOptions	The arguments that are passed by default as parameters to the Orbix E2A IDL compiler.	
CobolExtension <sup>a</sup>	Extension for the server mainline source code file on Windows NT.	cbl
ImplementationExtension <sup>a</sup>	Extension for the server implementation source code file on Windows NT. You should copy this to a file with a <code>.cbl</code> extension, to avoid overwriting any subsequent changes if you run the Orbix E2A IDL compiler again.	xxx
CopybookExtension <sup>a</sup>	Extension for COBOL copybooks on Windows NT.	cpy
MainCopybookSuffix	Suffix for the main copybook.	
RuntimeCopybookSuffix	Suffix for the runtime copybook.	x

**Table 24:** *COBOL Configuration Settings (Sheet 2 of 2)*

Setting Name	Description	Default
SelectCopybookSuffix	Suffix for the select copybook.	D
ImplementationSuffix	Suffix for the server implementation source code member.	S
ServerSuffix	Suffix for the server mainline source code member.	SV

a. This is specific to Windows NT. It is ignored on OS/390.

The last five settings in [Table 24](#) are not listed by default in `orbixhlq.CONFIG(IDL)`. If you want to change the generated member suffixes from the default values shown in [Table 24](#), you must manually enter the relevant setting name and its corresponding value.

---

# Adapter Mapping Member Configuration Settings

---

## Overview

The `-mfa` plug-in allows the IDL compiler to generate IMS or CICS adapter mapping members from IDL. The `orbixhlg.CONFIG(IDL)` member contains configuration settings relating to the generation of IMS or CICS adapter mapping members.

**Note:** See the *IMS Adapter Administrator's Guide* or *CICS Adapter Administrator's Guide* for more details about adapter mapping members.

## Configuration settings

The IMS or CICS adapter mapping member configuration is listed under `MFAMappings` as follows:

```
MFAMappings
{
    Switch = "mfa";
    ShlibName = "ORXBMFA";
    ShlibMajorVersion = "x";
    IsDefault = "NO";
    PresetOptions = "";

    # Mapping file includes extensions
    # The Default is .map and none for OS/390.
    # MFAMappingExtension = "";
    # The default suffix is A.
    # MFAMappingSuffix = "";
};
```

## Mandatory settings

The first three of the preceding settings are mandatory and must not be altered. They inform the Orbix E2A 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

---

---



# API Reference

*This chapter summarizes the API functions that are defined for the Orbix E2A COBOL runtime, in pseudo-code. It explains how to use each function, with an example of how to call it from COBOL.*

**In this chapter**

This chapter discusses the following topics:

<a href="#">API Reference Summary</a>	<a href="#">page 264</a>
<a href="#">API Reference Details</a>	<a href="#">page 268</a>
<a href="#">Deprecated APIs</a>	<a href="#">page 381</a>

**Note:** All parameters are passed by reference to COBOL APIs.

---

# API Reference Summary

---

## Introduction

This section provides a summary of the available API functions, in alphabetic order. See [“API Reference Details” on page 268](#) for more details of each function.

---

## Summary listing

```

ANYFREE(inout POINTER any-pointer)
// Frees memory allocated to an any.

ANYGET(in POINTER any-pointer,
       out buffer any-data-buffer)
// Extracts data out of an any.

ANYSET(inout POINTER any-pointer,
       in buffer any-data-buffer)
// Inserts data into an any.

COAERR(in buffer user-exception-buffer)
// Allows a COBOL server to raise a user exception for an
// operation.

COAGET(in buffer operation-buffer)
// Marshals in and inout arguments for an operation on the server
// side from an incoming request.

COAPUT(out buffer operation-buffer)
// Marshals return, out, and inout arguments for an operation on
// the server side from an incoming request.

COAREQ(in buffer request-details)
// Provides current request information

COARUN
// Indicates the server is ready to accept requests.

MEMALLOC(in 9(09) BINARY memory-size,
         out POINTER memory-pointer)
// Allocates memory at runtime from the program heap.

MEMFREE(inout POINTER memory-pointer)
// Frees dynamically allocated memory.

```

```

OBJDUP(in POINTER object-reference,
       out POINTER duplicate-obj-ref)
// Duplicates an object reference.

OBJGETID(in POINTER object-reference,
         out X(nn) object-id,
         in 9(09) BINARY object-id-length)
// Retrieves the object ID from an object reference.

OBJNEW(in X(nn) server-name,
       in X(nn) interface-name,
       in X(nn) object-id,
       out POINTER object-reference)
// Creates a unique object reference.

OBJREL(inout POINTER object-reference)
// Releases an object reference.

OBJRIR(in X(nn) desired-service,
       out POINTER object-reference)
// Returns an object reference to an object through which a
// service such as the Naming Service can be used.

OBJTOSTR(in POINTER object-reference,
         out POINTER object-string)
// Returns a stringified interoperable object reference (IOR)
// from a valid object reference.

ORBARGS(in X(nn) argument-string,
        in 9(09) BINARY argument-string-length,
        in X(nn) orb-name,
        in 9(09) BINARY orb-name-length)
// Initializes a client or server connection to an ORB.

ORBEXEC(in POINTER object-reference,
        in X(nn) operation-name,
        inout buffer operation-buffer,
        inout buffer user-exception-buffer)
// Invokes an operation on the specified object.

ORBHOST(in 9(09) BINARY hostname-length,
        out X(nn) hostname)
// Returns the hostname of the server

ORBREG(in buffer interface-description)
// Describes an IDL interface to the COBOL runtime.

```

```

ORBSRVR(in X(nn) server-name,
        in 9(09) BINARY server-name-length)
// Sets the server name for the current server process.

ORBSTAT(in buffer status-buffer)
// Registers the status information block.

ORBTIME(in 9(04) BINARY timeout-type
        in 9(09) BINARY timeout-value)
// Used by clients for setting the call timeout.
// Used by servers for setting the event timeout.

SEQALLOC(in 9(09) BINARY sequence-size,
         in X(nn) typecode-key,
         in 9(09) BINARY typecode-key-length,
         inout buffer sequence-control-data)
// Allocates memory for an unbounded sequence

SEQDUP(in buffer sequence-control-data,
       out buffer dupl-seq-control-data)
// Duplicates an unbounded sequence control block.

SEQFREE(inout buffer sequence-control-data)
// Frees the memory allocated to an unbounded sequence.

SEQGET(in buffer sequence-control-data,
       in 9(09) BINARY element-number,
       out buffer sequence-data)
// Retrieves the specified element from an unbounded sequence.

SEQSET(out buffer sequence-control-data,
       in 9(09) BINARY element-number,
       in buffer sequence-data)
// Places the specified data into the specified element of an
// unbounded sequence.

STRFREE(in POINTER string-pointer)
// Frees the memory allocated to a bounded string.

STRGET(in POINTER string-pointer,
       in 9(09) BINARY string-length,
       out X(nn) string)
// Copies the contents of an unbounded string to a bounded string.

STRLEN(in POINTER string-pointer,
       out 9(09) BINARY string-length)
// Returns the actual length of an unbounded string.

```

```

STRSET(out POINTER string-pointer,
       in 9(09) BINARY string-length,
       in X(nn) string)
// Creates a dynamic string from a PIC X(n) data item

STRSETP(out POINTER string-pointer,
        in 9(09) BINARY string-length,
        in X(nn) string)
// Creates a dynamic string from a PIC X(n) data item.

STRTOOBJ(in POINTER object-string,
         out POINTER object-reference)
// Creates an object reference from an interoperable object
// reference (IOR).

TYPEGET(inout POINTER any-pointer,
        in 9(09) BINARY typecode-key-length,
        out X(nn) typecode-key)
// Extracts the type name from an any.

TYPESET(inout POINTER any-pointer,
        in 9(09) BINARY typecode-key-length,
        in X(nn) typecode-key)
// Sets the type name of an any.

WSTRFREE(in POINTER string-pointer)
// Frees the memory allocated to a bounded wide string.

WSTRGET(in POINTER string-pointer,
        in 9(09) BINARY string-length,
        out G(nn) string)
// Copies the contents of an unbounded wide string to a bounded
// wide string.

WSTRLEN(in POINTER string-pointer,
        out 9(09) BINARY string-length)
// Returns the actual length of an unbounded wide string.

WSTRSET(out POINTER string-pointer,
        in 9(09) BINARY string-length
        in G(nn) string)
// Creates a dynamic wide string from a PIC G(n) data item

WSTRSETP(out POINTER string-pointer,
         in 9(09) BINARY string-length,
         in G(nn) string)
// Creates a dynamic wide string from a PIC G(n) data item.

```

---

# API Reference Details

---

**Introduction**

This section provides details of each available API function, in alphabetic order.

---

**In this section**

This section discusses the following topics:

<a href="#">ANYFREE</a>	<a href="#">page 270</a>
<a href="#">ANYGET</a>	<a href="#">page 272</a>
<a href="#">ANYSET</a>	<a href="#">page 274</a>
<a href="#">COAERR</a>	<a href="#">page 277</a>
<a href="#">COAGET</a>	<a href="#">page 281</a>
<a href="#">COAPUT</a>	<a href="#">page 286</a>
<a href="#">COAREQ</a>	<a href="#">page 292</a>
<a href="#">COARUN</a>	<a href="#">page 297</a>
<a href="#">MEMALLOC</a>	<a href="#">page 298</a>
<a href="#">MEMFREE</a>	<a href="#">page 300</a>
<a href="#">OBJDUP</a>	<a href="#">page 301</a>
<a href="#">OBJGETID</a>	<a href="#">page 303</a>
<a href="#">OBJNEW</a>	<a href="#">page 305</a>
<a href="#">OBJREL</a>	<a href="#">page 308</a>
<a href="#">OBJRIR</a>	<a href="#">page 310</a>
<a href="#">OBJTOSTR</a>	<a href="#">page 312</a>
<a href="#">ORBARGS</a>	<a href="#">page 314</a>
<a href="#">ORBEXEC</a>	<a href="#">page 317</a>

<a href="#">ORBHOST</a>	<a href="#">page 323</a>
<a href="#">ORBREG</a>	<a href="#">page 325</a>
<a href="#">ORBSRVR</a>	<a href="#">page 328</a>
<a href="#">ORBSTAT</a>	<a href="#">page 329</a>
<a href="#">ORBTIME</a>	<a href="#">page 333</a>
<a href="#">SEQALLOC</a>	<a href="#">page 335</a>
<a href="#">SEQDUP</a>	<a href="#">page 339</a>
<a href="#">SEQFREE</a>	<a href="#">page 344</a>
<a href="#">SEQGET</a>	<a href="#">page 347</a>
<a href="#">SEQSET</a>	<a href="#">page 350</a>
<a href="#">STRFREE</a>	<a href="#">page 355</a>
<a href="#">STRGET</a>	<a href="#">page 357</a>
<a href="#">STRLEN</a>	<a href="#">page 360</a>
<a href="#">STRSET</a>	<a href="#">page 362</a>
<a href="#">STRSETP</a>	<a href="#">page 365</a>
<a href="#">STRTOOBJ</a>	<a href="#">page 367</a>
<a href="#">TYPEGET</a>	<a href="#">page 369</a>
<a href="#">TYPESET</a>	<a href="#">page 371</a>
<a href="#">WSTRFREE</a>	<a href="#">page 373</a>
<a href="#">WSTRGET</a>	<a href="#">page 374</a>
<a href="#">WSTRLEN</a>	<a href="#">page 375</a>
<a href="#">WSTRSET</a>	<a href="#">page 376</a>
<a href="#">WSTRSETP</a>	<a href="#">page 377</a>
<a href="#">CHECK-STATUS</a>	<a href="#">page 378</a>

---

# ANYFREE

## Synopsis

---

```
ANYFREE(inout POINTER 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:

<code>any-pointer</code>	This is an <code>inout</code> parameter that is a pointer to the address in memory where the <code>any</code> is stored.
--------------------------	--

---

## Example

The example can be broken down as follows:

1. Consider the following IDL:

```
//IDL  
interface sample {  
    attribute any myany;  
};
```



2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

```
01 SAMPLE-MYANY-ARGS.
03 RESULT                                POINTER
                                         VALUE NULL.
```

3. The following is an example of how to use ANYFREE in your client or server program:

```
...
PROCEDURE DIVISION.
    CALL "ANYFREE" USING RESULT OF SAMPLE-MYANY-ARGS.
...
```

#### See also

- [“ANYSET” on page 274.](#)
- [“TYPESET” on page 371.](#)
- [“The any Type and Memory Management” on page 397.](#)

---

# ANYGET

## Synopsis

---

```
ANYGET(in POINTER any-pointer,  
       out buffer 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:

<code>any-pointer</code>	This is an <code>inout</code> parameter that is a pointer to the address in memory where the <code>any</code> is stored.
<code>any-data-buffer</code>	This is an <code>out</code> parameter that can be of any valid COBOL type. It is used to store the value extracted from the <code>any</code> .

---

## Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface sample {  
    attribute any myany;  
};
```

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

```

01 SAMPLE-MYANY-ARGS.
    03 RESULT                                POINTER
                                           VALUE NULL.
...
01 EXAMPLE-TYPE
    COPY CORBATYP.
    88 SAMPLE                                VALUE "IDL:sample:1.0".
01 EXAMPLE-TYPE-LENGTH                      PICTURE S9(09) BINARY
                                           VALUE 22.

```

3. The following is an example of how to use ANYSET in a client or server program:

```

WORKING-STORAGE SECTION.
    01 WS-DATA                                PIC S9(10) VALUE 0.

CALL "TYPEGET" USING RESULT OF SAMPLE-MYANY-ARGS
                        EXAMPLE-TYPE-LENGTH
                        EXAMPLE-TYPE.

SET WS-TYPEGET TO TRUE.
PERFORM CHECK-STATUS.
* validate typecode
EVALUATE TRUE
    WHEN CORBA-TYPE-LONG
* retrieve the ANY CORBA::Short value
    CALL "ANYGET" USING RESULT OF SAMPLE-MYANY-ARGS
                        WS-DATA
    SET WS-ANYGET TO TRUE
    PERFORM CHECK-STATUS
    DISPLAY "ANY value equals " WS-DATA.
    WHEN OTHER
        DISPLAY "Wrong typecode received, expected a LONG
                typecode"
END-EVALUTE.

```

See also

[“ANYSET” on page 274.](#)

---

# ANYSET

Synopsis	<pre>ANYSET(inout POINTER any-pointer,        in <i>buffer</i> any-data-buffer) // Inserts data into an any.</pre>				
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 <i>any</i> . ANYSET allocates memory that is required to store the value of the <i>any</i> . You must call TYPESET before calling ANYSET, to set the typecode of the <i>any</i> . Ensure that this typecode matches the type of the data being copied to the <i>any</i> .				
Parameters	<p>The parameters for ANYSET can be described as follows:</p> <table><tr><td><i>any-pointer</i></td><td>This is an <i>inout</i> parameter that is a pointer to the address in memory where the <i>any</i> is stored.</td></tr><tr><td><i>any-data-buffer</i></td><td>This is an <i>in</i> parameter that can be of any valid COBOL type. It contains the value to be copied to the <i>any</i>.</td></tr></table>	<i>any-pointer</i>	This is an <i>inout</i> parameter that is a pointer to the address in memory where the <i>any</i> is stored.	<i>any-data-buffer</i>	This is an <i>in</i> parameter that can be of any valid COBOL type. It contains the value to be copied to the <i>any</i> .
<i>any-pointer</i>	This is an <i>inout</i> parameter that is a pointer to the address in memory where the <i>any</i> is stored.				
<i>any-data-buffer</i>	This is an <i>in</i> parameter that can be of any valid COBOL type. It contains the value to be copied to the <i>any</i> .				
Example	<p>The example can be broken down as follows:</p> <ol style="list-style-type: none"><li>1. Consider the following IDL:</li></ol> <pre>interface sample {     attribute any myany; };</pre>				

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

```

01 SAMPLE-MYANY-ARGS.
    03 RESULT                                POINTER
                                           VALUE NULL.
...
01 EXAMPLE-TYPE                                PICTURE X(15).
    COPY CORBATYP.
    88 SAMPLE                                VALUE "IDL:sample:1.0".
01 EXAMPLE-TYPE-LENGTH                        PICTURE S9(09) BINARY
                                           VALUE 22.

```

3. The following is an example of how to use ANYSET in a client or server program:

```

WORKING-STORAGE SECTION.
01 WS-DATA                                PIC S9(10) VALUE 100.

PROCEDURE DIVISION.
...
* Set the ANY typecode to be a CORBA::Long
  SET CORBA-TYPE-LONG TO TRUE.
  CALL "TYPESET" USING RESULT OF
                                SAMPLE-MYANY-ARGS
                                EXAMPLE-TYPE-LENGTH
                                EXAMPLE-TYPE.

  SET WS-TYPESET TO TRUE.
  PERFORM CHECK-STATUS.
* Set the ANY value to 100
  CALL "ANYSET" USING RESULT OF SAMPLE-MYANY-ARGS
                                WS-DATA.

  SET WS-TYPESET TO TRUE.
  PERFORM CHECK-STATUS.

```

## 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 272.](#)
  - [“TYPESET” on page 371.](#)
  - [“The any Type and Memory Management” on page 397.](#)

---

# COAERR

## Synopsis

---

```
COAERR(in buffer user-exception-buffer)
// Allows a COBOL server to raise a user exception for an
// operation.
```

---

## Usage

---

Server-specific.

---

## Description

The `COAERR` function allows a COBOL 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 `ORBEXEC` 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 `COAERR` instead of `COAPUT` in this instance, and this informs the client that a user exception has been raised. Refer to the [“Memory Handling” on page 383](#) for more details. Calling `COAERR` 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` parameter passed to `ORBEXEC` is equal to zero after the call. Refer to [“ORBEXEC” on page 317](#) for an example of how a COBOL client determines if a user exception has been raised.

---

## Parameters

---

The parameter for `COAERR` can be described as follows:

<code>user-exception-buffer</code>	This is an <code>in</code> parameter that contains the COBOL representation of the user exceptions that the operation supports, as defined in the <code>idlmembername</code> copybook generated by the Orbix E2A IDL compiler. If the IDL operation supports no user exceptions, a dummy buffer is generated—this dummy buffer is not populated on the server side, and it is only used as the fourth (in this case, dummy) parameter to <code>ORBEXEC</code> .
------------------------------------	---

**Example**

The example can be broken down as follows:

1. Consider the following IDL:

```
//IDL
interface sample {
    typedef string<10> Aboundedstring;
    exception MyException { Aboundedstring except_str; };
    Aboundedstring myoperation(in Aboundedstring instr,
        inout Aboundedstring inoutstr,
        out Aboundedstring outstr)
        raises (MyException);
};
```

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

**Example 22: The *idlmembername* Copybook (Sheet 1 of 2)**

```
*****
* Operation:      myoperation
* Mapped name:    myoperation
* Arguments:      <in> sample/Aboundedstring instr
*                 <inout> sample/Aboundedstring inoutstr
*                 <out> sample/Aboundedstring outstr
* Returns:        sample/Aboundedstring
* User Exceptions: sample/MyException
*****
* operation-buffer
01 SAMPLE-MYOPERATION-ARGS.
   03 INSTR                      PICTURE X(10).
   03 INOUTSTR                   PICTURE X(10).
   03 OUTSTR                     PICTURE X(10).
   03 RESULT                    PICTURE X(10).
*****
COPY EXAMPLX.
*****

*****
*
* Operation List section
* This lists the operations and attributes which an
* interface supports
```



**Example 22: The idlmembername Copybook (Sheet 2 of 2)**

```

*
*****
* The operation-name and its corresponding 88 level entry
01 SAMPLE-OPERATION                                PICTURE X(27).
    88 SAMPLE-MYOPERATION                            VALUE
        "myoperation:IDL:sample:1.0".
01 SAMPLE-OPERATION-LENGTH                          PICTURE 9(09)
                                                    BINARY VALUE 27.

*****
*
* Typecode section
* This contains CDR encodings of necessary typecodes.
*
*****

01 EXAMPLE-TYPE                                    PICTURE X(29).
    COPY CORBATYP.
    88 SAMPLE-ABOUNDEDSTRING                        VALUE
        "IDL:sample/Aboundedstring:1.0".
01 EXAMPLE-TYPE-LENGTH                              PICTURE S9(09)
                                                    BINARY VALUE 29.
*****
* User exception block
*****
01 EX-SAMPLE-MYEXCEPTION                            PICTURE X(26)
                                                    VALUE
        "IDL:sample/MyException:1.0".
01 EX-SAMPLE-MYEXCEPTION-LENGTH                      PICTURE 9(09)
                                                    BINARY VALUE 26.

* user-exception-buffer

01 EXAMPLE-USER-EXCEPTIONS.
    03 EXCEPTION-ID                                POINTER
                                                    VALUE NULL.

    03 D                                            PICTURE 9(10) BINARY
                                                    VALUE 0.

        88 D-NO-USEREXCEPTION                        VALUE 0.
        88 D-SAMPLE-MYEXCEPTION                      VALUE 1.
    03 U                                            PICTURE X(10)
                                                    VALUE LOW-VALUES.

    03 EXCEPTION-SAMPLE-MYEXCEPTION                REDEFINES U.
    05 EXCEPT-STR                                PICTURE X(10).

```

3. The following is an example of the server implementation code for the `myoperation` operation:

```
DO-SAMPLE-MYOPERATION.
    SET D-NO-USEREXCEPTION TO TRUE.
    CALL "COAGET" USING SAMPLE-MYOPERATION-ARGS.
    SET WS-COAGET TO TRUE.
    PERFORM CHECK-STATUS.

    * Assuming some error has occurred in the application
      IF APPLICATION-ERROR
    * Raise the appropriate user exception
      SET D-SAMPLE-MYEXCEPTION TO TRUE

    * Populate the values of the exception to be passed back to
    * the client
      MOVE "FATAL ERROR " TO EXCEPT-STR
      OF EXAMPLE-USER-EXCEPTIONS
      CALL "COAERR" USING EXAMPLE-USER-EXCEPTIONS
      SET WS-COAERR TO TRUE
      PERFORM CHECK-STATUS
    ELSE
    *all okay pass back the out/inout/return parameters.
      CALL "COAPUT" USING SAMPLE-MYOPERATION-ARGS
      SET WS-COAPUT TO TRUE
      PERFORM CHECK-STATUS
    END-IF.
```

## 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

- [“COAGET” on page 281.](#)
- [“COAPUT” on page 286.](#)
- [“ORBEXEC” on page 317.](#)
- The `BANK` demonstration in `orbixhlq.DEMOS.COBOI.SRC` for a complete example of how to use `COAERR`.

---

# COAGET

## Synopsis

---

```
COAGET(in buffer 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 `COAGET` and end with a call to `COAPUT`. Even if the operation takes no parameters and has no return value, you must still call `COAGET` and `COAPUT` and, in such cases, pass a dummy `PIC X(1)` data item, which the Orbix E2A IDL compiler generates for such cases.

`COAGET` copies the incoming operation's argument values into the complete COBOL operation parameter buffer that is supplied. This buffer is generated automatically by the Orbix E2A IDL compiler. Only `in` and `inout` values in this structure are populated by this call.

The Orbix E2A IDL compiler generates the call for `COAGET` in the *idlmembername*`S` source module (where *idlmembername* represents the name of the IDL member that contains the IDL definitions) for each attribute and operation defined in the IDL.

---

## Parameters

The parameter for `COAGET` can be described as follows:

<code>operation-buffer</code>	This is an <code>in</code> parameter that contains a COBOL <code>01</code> level data item representing the data types that the operation supports.
-------------------------------	---

**Example**

The example can be broken down as follows:

1. Consider the following IDL:

```
interface sample {
    typedef string<10> Aboundedstring;
    exception MyException { Aboundedstring except_str; };
    Aboundedstring myoperation(in Aboundedstring instr,
        inout Aboundedstring inoutstr,
        out Aboundedstring outstr)
        raises (MyException);
};
```

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

**Example 23:** *The idlmembername Copybook (Sheet 1 of 2)*

```
*****
* Operation:      myoperation
* Mapped name:    myoperation
* Arguments:      <in> sample/Aboundedstring instr
*                <inout> sample/Aboundedstring inoutstr
*                <out> sample/Aboundedstring outstr
* Returns:        sample/Aboundedstring
* User Exceptions: sample/MyException
*****
* operation-buffer
01 SAMPLE-MYOPERATION-ARGS.
   03 INSTR                      PICTURE X(10).
   03 INOUTSTR                   PICTURE X(10).
   03 OUTSTR                     PICTURE X(10).
   03 RESULT                    PICTURE X(10).
*****
COPY EXAMPLX.
*****
*
* Operation List section
* This lists the operations and attributes which an
* interface supports
*
*****
```

**Example 23:** *The idlmembername Copybook (Sheet 2 of 2)*

```

* The operation-name and its corresponding 88 level entry
01 SAMPLE-OPERATION                                PICTURE X(27).
    88 SAMPLE-MYOPERATION                            VALUE
        "myoperation:IDL:sample:1.0".
01 SAMPLE-OPERATION-LENGTH                          PICTURE 9(09)
                                                    BINARY VALUE 27.

*****
*
* Typecode section
* This contains CDR encodings of necessary typecodes.
*
*****

01 EXAMPLE-TYPE                                    PICTURE X(29).
    COPY CORBATYP.
    88 SAMPLE-ABOUNDEDSTRING                        VALUE
        "IDL:sample/Aboundedstring:1.0".
01 EXAMPLE-TYPE-LENGTH                            PICTURE S9(09)
                                                    BINARY VALUE 29.
*****

* User exception block
*****
01 EX-SAMPLE-MYEXCEPTION                          PICTURE X(26)
                                                    VALUE
        "IDL:sample/MyException:1.0".
01 EX-SAMPLE-MYEXCEPTION-LENGTH                    PICTURE 9(09)
                                                    BINARY VALUE 26.

* user-exception-buffer

01 EXAMPLE-USER-EXCEPTIONS.
    03 EXCEPTION-ID                                POINTER
                                                    VALUE NULL.
    03 D                                            PICTURE 9(10)
                                                    BINARY VALUE 0.
    88 D-NO-USEREXCEPTION                          VALUE 0.
    88 D-SAMPLE-MYEXCEPTION                        VALUE 1.
    03 U                                            PICTURE X(10)
                                                    VALUE LOW-VALUES.
    03 EXCEPTION-SAMPLE-MYEXCEPTION                REDEFINES U.
    05 EXCEPT-STR                                PICTURE X(10).

```

3. The following is an example of the server implementation code for the `myoperation` operation, which is generated in the `idlmemberNames` source member when you specify the `-z` argument with the Orbix E2A IDL compiler:

```
DO-SAMPLE-MYOPERATION.
    SET D-NO-USEREXCEPTION TO TRUE.
    CALL "COAGET" USING SAMPLE-MYOPERATION-ARGS.
    SET WS-COAGET TO TRUE.
    PERFORM CHECK-STATUS.

* TODO: Add your operation specific code here

    EVALUATE TRUE
    WHEN D-NO-USEREXCEPTION
    CALL "COAPUT" USING SAMPLE-MYOPERATION-ARGS
    SET WS-COAPUT TO TRUE
    PERFORM CHECK-STATUS
    END-EVALUATE.
```

4. The following is an example of a modified version of the code in point 3 for the `myoperation` operation:

When changed for this operation can look like this  
Sample server implementation for `myoperation`

```
DO-SAMPLE-MYOPERATION.
    SET D-NO-USEREXCEPTION TO TRUE.
    CALL "COAGET" USING SAMPLE-MYOPERATION-ARGS.
    SET WS-COAGET TO TRUE.
* Display what the client passed in
    DISPLAY "In parameter value equals "
    INSTR OF SAMPLE-MYOPERATION-ARGS.
    DISPLAY "Inout parameter value equals "
    INOUTSTR OF SAMPLE-MYOPERATION-ARGS.

*Now must populate the inout/out/return parameters if
*applicable. See COAPUT for example.
    EVALUATE TRUE
    WHEN D-NO-USEREXCEPTION
    CALL "COAPUT" USING SAMPLE-MYOPERATION-ARGS
    SET WS-COAPUT TO TRUE
    PERFORM CHECK-STATUS
    END-EVALUATE.
```

## 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.

A `CORBA::DATA_CONVERSION::VALUE_OUT_OF_RANGE` exception is raised if the value is determined to be out of range when marshalling a `long`, `short`, `unsigned short`, `unsigned long long long`, or `unsigned long long` type.

---

## See also

- [“COAERR” on page 277](#).
- [“ORBEXEC” on page 317](#).

---

# COAPUT

Synopsis	<pre>COAPUT(out <i>buffer</i> operation-buffer) // Marshals return, out, and inout arguments for an operation on // the server side from an incoming request.</pre>		
Usage	Server-specific.		
Description	<p>Each operation implementation must begin with a call to <code>COAGET</code> and end with a call to <code>COAPUT</code>. The <code>COAPUT</code> function copies the operation's outgoing argument values from the complete COBOL operation parameter buffer passed to it. This buffer is generated automatically by the Orbix E2A IDL compiler. Only <code>inout</code>, <code>out</code>, and the <code>result out</code> item are populated by this call.</p> <p>You must ensure that all <code>inout</code>, <code>out</code>, and <code>result</code> values are correctly allocated (for dynamic types) and populated. If a user exception has been raised before calling <code>COAPUT</code>, no <code>inout</code>, <code>out</code>, or <code>result</code> parameters are marshalled, and nothing is returned in such cases. If a user exception has been raised, <code>COAERR</code> must be called instead of <code>COAPUT</code>, and no <code>inout</code>, <code>out</code>, or <code>result</code> parameters are marshalled. Refer to <a href="#">“COAERR” on page 277</a> for more details.</p> <p>The Orbix E2A IDL compiler generates the call for <code>COAPUT</code> in the <code>idlmembernameS</code> source module for each attribute and operation defined in the IDL.</p>		
Parameters	<p>The parameter for <code>COAPUT</code> can be described as follows:</p> <table><tr><td><code>operation-buffer</code></td><td>This is an <code>out</code> parameter that contains a COBOL 01 level data item representing the data types that the operation supports.</td></tr></table>	<code>operation-buffer</code>	This is an <code>out</code> parameter that contains a COBOL 01 level data item representing the data types that the operation supports.
<code>operation-buffer</code>	This is an <code>out</code> parameter that contains a COBOL 01 level data item representing the data types that the operation supports.		



**Example**

The example can be broken down as follows:

1. Consider the following IDL:

```
interface sample {
    typedef string<10> Aboundedstring;
    exception MyException { Aboundedstring except_str; };
    Aboundedstring myoperation(in Aboundedstring instr,
        inout Aboundedstring inoutstr,
        out Aboundedstring outstr)
        raises (MyException);
};
```

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

**Example 24:** *The idlmembername Copybook (Sheet 1 of 2)*

```
*****
* Operation:      myoperation
* Mapped name:    myoperation
* Arguments:      <in> sample/Aboundedstring instr
*                 <inout> sample/Aboundedstring inoutstr
*                 <out> sample/Aboundedstring outstr
* Returns:        sample/Aboundedstring
* User Exceptions: sample/MyException
*****
* operation-buffer
01 SAMPLE-MYOPERATION-ARGS.
   03 INSTR                      PICTURE X(10).
   03 INOUTSTR                   PICTURE X(10).
   03 OUTSTR                     PICTURE X(10).
   03 RESULT                    PICTURE X(10).
*****
COPY EXAMPLX.
*****
*
* Operation List section
* This lists the operations and attributes which an
* interface supports
*
*****
```

**Example 24:** *The idlmembername Copybook (Sheet 2 of 2)*

```

* The operation-name and its corresponding 88 level entry
01 SAMPLE-OPERATION                                PICTURE X(27).
    88 SAMPLE-MYOPERATION                            VALUE
        "myoperation:IDL:sample:1.0".
01 SAMPLE-OPERATION-LENGTH                          PICTURE 9(09)
                                                    BINARY VALUE 27.

*****
*
* Typecode section
* This contains CDR encodings of necessary typecodes.
*
*****

01 EXAMPLE-TYPE                                    PICTURE X(29).
    COPY CORBATYP.
    88 SAMPLE-ABOUNDEDSTRING                        VALUE
        "IDL:sample/Aboundedstring:1.0".
01 EXAMPLE-TYPE-LENGTH                            PICTURE S9(09)
                                                    BINARY VALUE 29.

*****

* User exception block
*****

01 EX-SAMPLE-MYEXCEPTION                          PICTURE X(26)
                                                    VALUE
        "IDL:sample/MyException:1.0".
01 EX-SAMPLE-MYEXCEPTION-LENGTH                    PICTURE 9(09)
                                                    BINARY VALUE 26.

* user exception buffer
01 EXAMPLE-USER-EXCEPTIONS.
    03 EXCEPTION-ID                                POINTER
                                                    VALUE NULL.
    03 D                                            PICTURE 9(10)
                                                    BINARY.
                                                    VALUE 0.
    88 D-NO-USEREXCEPTION                            VALUE 0.
    88 D-SAMPLE-MYEXCEPTION                        VALUE 1.
    03 U                                            PICTURE X(10)
                                                    VALUE LOW-VALUES.
    03 EXCEPTION-SAMPLE-MYEXCEPTION                REDEFINES U.
    05 EXCEPT-STR                                PICTURE X(10).

```

3. The following is an example of the server implementation code for the `myoperation` operation, which is generated in the `idlmembernameS` source member when you specify the `-z` argument with the Orbix E2A IDL compiler:

```
DO-SAMPLE-MYOPERATION.  
    SET D-NO-USEREXCEPTION TO TRUE.  
    CALL "COAGET" USING SAMPLE-MYOPERATION-ARGS.  
    SET WS-COAGET TO TRUE.  
    PERFORM CHECK-STATUS.  
  
* TODO: Add your operation specific code here  
  
    EVALUATE TRUE  
    WHEN D-NO-USEREXCEPTION  
    CALL "COAPUT" USING SAMPLE-MYOPERATION-ARGS  
    SET WS-COAPUT TO TRUE  
    PERFORM CHECK-STATUS  
    END-EVALUATE.
```

4. The following is an example of a modified version of the code in point 3 for the `myoperation` operation

```
When changed for this operation can look like this
Sample server implementation for myoperation

DO-SAMPLE-MYOPERATION.
    SET D-NO-USEREXCEPTION TO TRUE.
    CALL "COAGET" USING SAMPLE-MYOPERATION-ARGS.
    SET WS-COAGET TO TRUE.
* Display what the client passed in
    DISPLAY "In parameter value equals "
    INSTR OF SAMPLE-MYOPERATION-ARGS.
    DISPLAY "Inout parameter value equals "
    INOUTSTR OF SAMPLE-MYOPERATION-ARGS.

*Now must populate the inout/out/return parameters if
*applicable
    MOVE "Client" TO INOUTSTR OF SAMPLE-MYOPERATION-ARGS.
    MOVE "xxxxx" TO OUTSTR OF SAMPLE-MYOPERATION-ARGS.
    MOVE "YYYYY" TO RESULT OF SAMPLE-MYOPERATION-ARGS.

    EVALUATE TRUE
    WHEN D-NO-USEREXCEPTION
    CALL "COAPUT" USING SAMPLE-MYOPERATION-ARGS
    SET WS-COAPUT TO TRUE
    PERFORM CHECK-STATUS
    END-EVALUATE.
```

## 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.

A `CORBA::DATA_CONVERSION::VALUE_OUT_OF_RANGE` exception is raised if the value is determined to be out of range when marshalling a long, short, unsigned short, unsigned long long long, or unsigned long long type.

**See also**

- 
- [“COAERR” on page 277.](#)
  - [“ORBEXEC” on page 317.](#)

# COAREQ

Synopsis

COAREQ(in *buffer* request-details)  
// Provides current request information

Usage

Server-specific.

Description

The server implementation program calls COAREQ to extract the relevant information about the current request. COAREQ provides information about the current invocation request in a request information buffer, which is defined as follows in the supplied CORBA copybook:

```
01 REQUEST-INFO.  
  03 INTERFACE-NAME      USAGE IS POINTER VALUE NULL.  
  03 OPERATION-NAME      USAGE IS POINTER VALUE NULL.  
  03 PRINCIPAL            USAGE IS POINTER VALUE NULL.  
  03 TARGET               USAGE IS POINTER VALUE NULL.
```

In the preceding structure, the first three data items are unbounded CORBA character strings. You can use the STRGET function to copy the values of these strings to COBOL bounded string data items. The TARGET item in the preceding structure is the COBOL object reference for the operation invocation. After COAREQ 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 COAREQ only once for each operation invocation. It must be called after a request has been dispatched to a server, and before any calls are made to access the parameter values. Supplied code is generated in the *idlmembernameS* source module by the Orbix E2A IDL compiler when you specify the *-z* argument. Ensure that the COBOL bounded string and the length fields are large enough to retrieve the data from the REQUEST-INFO pointers.

---

**Parameters**

The parameter for `COAREQ` can be described as follows:

`request-details`      This is an `in` parameter that contains a COBOL 01 level data item representing the current request.

---

**Example**

The example can be broken down as follows:

1. Consider the following IDL:

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

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the `idlmembername` copybook (where `idlmembername` represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

**Example 25:** *The `idlmembername` Copybook (Sheet 1 of 2)*

```
*****
* Operation:      call_me
* Mapped name:    call_me
* Arguments:      None
* Returns:        void
* User Exceptions: none
*****
01 SIMPLE-SIMPLEOBJECT-70FE-ARGS.
   03 FILLER                                           PICTURE X(01).
*****
COPY SIMPLEX.
*****
*
* Operation List section
* This lists the operations and attributes which an
* interface supports
*
```

**Example 25:** *The idlmembername Copybook (Sheet 2 of 2)*

```

*****
01 SIMPLE-SIMPLEOBJECT-OPERATION          PICTURE X(36).
   88 SIMPLE-SIMPLEOBJECT-CALL-ME          VALUE
      "call_me:IDL:Simple/SimpleObject:1.0".
01 SIMPLE-S-3497-OPERATION-LENGTH          PICTURE 9(09)
                                           BINARY VALUE 36.

*****
*
* Typecode section
* This contains CDR encodings of necessary typecodes.
*
*****
01 SIMPLE-TYPE                             PICTURE X(27).
   COPY CORBATYP.
   88 SIMPLE-SIMPLEOBJECT                  VALUE
      "IDL:Simple/SimpleObject:1.0".
01 SIMPLE-TYPE-LENGTH                     PICTURE S9(09)
                                           BINARY VALUE 27.

```

3. The following is an example of the server implementation code generated in the *idlmembernameS* server implementation member:

**Example 26:** *Part of the idlmembernameS Program (Sheet 1 of 2)*

```

WORKING-STORAGE SECTION
01 WS-INTERFACE-NAME                      PICTURE X(30).
01 WS-INTERFACE-NAME-LENGTH              PICTURE 9(09) BINARY
                                           VALUE 30.

PROCEDURE DIVISION.

    ENTRY "DISPATCH".

    CALL "COAREQ"      USING REQUEST-INFO.
    SET WS-COAREQ TO TRUE.
    PERFORM CHECK-STATUS.

    * Resolve the pointer reference to the interface name
    * which is the fully scoped interface name.
    * Note make sure it can handle the max interface name
    * length.
    CALL "STRGET"      USING INTERFACE-NAME
                                           WS-INTERFACE-NAME-LENGTH

```



**Example 26:** *Part of the idlmembernameS Program (Sheet 2 of 2)*

```

WS-INTERFACE-NAME.

SET WS-STRGET TO TRUE.
PERFORM CHECK-STATUS.

*****
* Interface(s)  evaluation:
*****
      MOVE SPACES TO SIMPLE-SIMPLEOBJECT-OPERATION.

      EVALUATE WS-INTERFACE-NAME
      WHEN 'IDL:Simple/SimpleObject:1.0'
* Resolve the pointer reference to the operation
* information
      CALL "STRGET" USING OPERATION-NAME
                           SIMPLE-S-3497-OPERATION-LENGTH
                           SIMPLE-SIMPLEOBJECT-OPERATION
      SET WS-STRGET TO TRUE
      PERFORM CHECK-STATUS
      DISPLAY  "Simple::" SIMPLE-SIMPLEOBJECT-OPERATION
              "invoked"
      END-EVALUATE.
COPY SIMPLED.

      GOBACK.
DO-SIMPLE-SIMPLEOBJECT-CALL-ME.
      CALL "COAGET"      USING SIMPLE-SIMPLEOBJECT-70FE-ARGS.
      SET WS-COAGET TO TRUE.
      PERFORM CHECK-STATUS.

      CALL "COAPUT"      USING SIMPLE-SIMPLEOBJECT-70FE-ARGS.
      SET WS-COAPUT TO TRUE.
      PERFORM CHECK-STATUS.

*****
* Check Errors Copybook
*****
      COPY CHKERRS.

```

**Note:** The COPY CHKERRS statement in the preceding example is used in batch programs. It is replaced with COPY CERRSMFA in IMS or CICS server programs.

## 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 `ORBSVR` is not called.

---

# COARUN

---

## Synopsis

```
COARUN
// Indicates the server is ready to accept requests.
```

---

## Usage

Server-specific.

---

## Description

The `COARUN` function indicates that a server is ready to start receiving client requests. It is equivalent to calling `ORB::run()` in C++. Refer to the *CORBA Programmer's Reference, C++* for more details about `ORB::run()`. There are no parameters required for calling `COARUN`. The call to `COARUN` is the final API call in your server mainline program.

---

## Parameters

`COARUN` takes no parameters.

---

## Example

The following is an example of how to use `COARUN` in your server mainline program:

```
DISPLAY "Giving control to the ORB to process requests".

CALL "COARUN".
SET WS-COARUN TO TRUE.
PERFORM CHECK-STATUS.
```

---

## Exceptions

A `CORBA::BAD_INV_ORDER::SERVER_NAME_NOT_SET` exception is raised if `ORBSVR` is not called.

---

# MEMALLOC

---

Synopsis	<pre>MEMALLOC(in 9(09) BINARY memory-size,          out POINTER memory-pointer) // Allocates memory at runtime from the program heap.</pre>				
Usage	Common to clients and servers.				
Description	<p>The <code>MEMALLOC</code> function allocates the specified number of bytes from the program heap at runtime, and returns a pointer to the start of this memory block.</p> <p><code>MEMALLOC</code> is used to allocate space for dynamic structures. However, it is recommended that you use <code>SEQALLOC</code> when allocating memory for sequences, because <code>SEQALLOC</code> can automatically determine the amount of memory required for sequences. Refer to <a href="#">“SEQALLOC” on page 335</a> for more details.</p>				
Parameters	<p>The parameters for <code>MEMALLOC</code> can be described as follows:</p> <table><tr><td><code>memory-size</code></td><td>This is an <code>in</code> parameter that specifies in bytes the amount of memory that is to be allocated.</td></tr><tr><td><code>memory-pointer</code></td><td>This is an <code>out</code> parameter that contains a pointer to the allocated memory block.</td></tr></table>	<code>memory-size</code>	This is an <code>in</code> parameter that specifies in bytes the amount of memory that is to be allocated.	<code>memory-pointer</code>	This is an <code>out</code> parameter that contains a pointer to the allocated memory block.
<code>memory-size</code>	This is an <code>in</code> parameter that specifies in bytes the amount of memory that is to be allocated.				
<code>memory-pointer</code>	This is an <code>out</code> parameter that contains a pointer to the allocated memory block.				
Exceptions	<p>A <code>CORBA::NO_MEMORY</code> exception is raised if there is not enough memory available to complete the request. In this case, the pointer will contain a null value.</p>				

---

**Example**

The following is an example of how to use `MEMALLOC` in a client or server program:

```
WORKING-STORAGE SECTION.  
  
01 WS-MEMORY-BLOCK                POINTER VALUE NULL.  
01 WS-MEMORY-BLOCK-SIZE          PICTURE 9(09) BINARY VALUE 30.  
  
PROCEDURE DIVISION.  
...  
* allocates 30 bytes of memory at runtime from the heap  
  CALL "MEMALLOC" USING WS-MEMORY-BLOCK-SIZE  
                        WS-MEMORY-BLOCK.
```

---

**See also**

- [“MEMFREE” on page 300.](#)
- [“Memory Handling” on page 383.](#)

---

# MEMFREE

Synopsis	<pre>MEMFREE(inout POINTER memory-pointer) // Frees dynamically allocated memory.</pre>		
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 MEMALLOC. Do not try to use this pointer after freeing it, because doing so might result in a runtime error.		
Parameters	<p>The parameter for MEMFREE can be described as follows:</p> <table><tr><td>memory-pointer</td><td>This is an inout parameter that contains a pointer to the allocated memory block.</td></tr></table>	memory-pointer	This is an inout parameter that contains a pointer to the allocated memory block.
memory-pointer	This is an inout parameter that contains a pointer to the allocated memory block.		
Example	<p>The following is an example of how to use MEMFREE in a client or server program:</p> <pre>WORKING-STORAGE SECTION. 01 WS-MEMORY-BLOCK                POINTER VALUE NULL.  PROCEDURE DIVISION.      ...      * Finished with the block of memory allocated by call to MEMALLOC     CALL "MEMFREE" USING WS-MEMORY-BLOCK.</pre>		
See also	<a href="#">“MEMALLOC” on page 298.</a>		

---

# OBJDUP

Synopsis	<pre>OBJDUP(in POINTER object-reference,        out POINTER duplicate-obj-ref) // Duplicates an object reference.</pre>				
Usage	Common to clients and servers.				
Description	<p>The <code>OBJDUP</code> 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 <code>CORBA::Object::_duplicate()</code> 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 <code>OBJDUP</code> to create a duplicate.</p>				
Parameters	<p>The parameters for <code>OBJDUP</code> can be described as follows:</p> <table><tr><td><code>object-reference</code></td><td>This is an <code>in</code> parameter that contains the valid object reference.</td></tr><tr><td><code>duplicate-obj-ref</code></td><td>This is an <code>out</code> parameter that contains the duplicate object reference.</td></tr></table>	<code>object-reference</code>	This is an <code>in</code> parameter that contains the valid object reference.	<code>duplicate-obj-ref</code>	This is an <code>out</code> parameter that contains the duplicate object reference.
<code>object-reference</code>	This is an <code>in</code> parameter that contains the valid object reference.				
<code>duplicate-obj-ref</code>	This is an <code>out</code> parameter that contains the duplicate object reference.				
Example	<p>The following is an example of how to use <code>OBJDUP</code> in a client or server program:</p>				

```
WORKING-STORAGE SECTION.  
01 WS-SIMPLE-SIMPLEOBJECT          POINTER VALUE NULL.  
01 WS-SIMPLE-SIMPLEOBJECT-COPY     POINTER VALUE NULL.  
  
PROCEDURE DIVISION.  
...  
* Note that the object reference will have been created,  
* for example, by a call to OBJNEW.  
    CALL "OBJDUP" USING WS-SIMPLE-SIMPLEOBJECT  
                        WS-SIMPLE-SIMPLEOBJECT-COPY.  
    SET WS-OBJDUP TO TRUE.  
    PERFORM CHECK-STATUS.
```





---

# OBJGETID

Synopsis	<pre>OBJGETID(in POINTER object-reference,          out X(nn) object-id,          in 9(09) BINARY object-id-length) // Retrieves the object ID from an object reference.</pre>						
Usage	Common to clients and servers.						
Description	The OBJGETID function retrieves the object ID string from an object reference. It is equivalent to calling POA::reference_to_id in C++.						
Parameters	<p>The parameters for OBJGETID can be described as follows:</p> <table><tr><td>object-reference</td><td>This is an in parameter that contains the valid object reference.</td></tr><tr><td>object-id</td><td>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.</td></tr><tr><td>object-id-length</td><td>This is an in parameter that specifies the length of the object name.</td></tr></table>	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.
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.						
Exceptions	<p>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.</p> <p>A CORBA::BAD_PARAM::INVALID_OBJECT_ID exception is raised if an Orbix 2.3 object reference is passed.</p> <p>A CORBA::BAD_INV_ORDER::SERVER_NAME_NOT_SET exception is raised if ORBSRV is not called.</p>						

**Example**

The following is an example of how to use `OBJGETID` in a client or server program:

```
WORKING-STORAGE SECTION.  
  
01 WS-OBJECT-IDENTIFIER-LEN    PICTURE 9(09) BINARY VALUE 0.  
01 WS-OBJECT-IDENTIFIER       PICTURE X(20) VALUE SPACES.  
01 WS-OBJECT                   POINTER VALUE NULL.  
  
PROCEDURE DIVISION.  
...  
* Note that the object reference will have been created, for  
* example, by a call to OBJNEW.  
  
    MOVE 20 TO WS-OBJECT-ID-LENGTH.  
    CALL "OBJGETID" USING WS-OBJECT  
                        WS-OBJECT-IDENTIFIER  
                        WS-OBJECT-IDENTIFIER-LEN.  
    SET WS-OBJGETID TO TRUE.  
    PERFORM CHECK-STATUS.  
  
    DISPLAY "Object identifier string equals "  
        WS-OBJECT-ID-STRING.
```

---

# OBJNEW

Synopsis	<pre>OBJNEW(in X(nn) server-name,        in X(nn) interface-name,        in X(nn) object-id,        out POINTER object-reference) // Creates a unique object reference.</pre>								
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	<p>The parameters for OBJNEW can be described as follows:</p> <table><tr><td>server-name</td><td>This is an <i>in</i> parameter that is a bounded string containing the server name. This must be the same as the value passed to ORBSRV. This string must be terminated by at least one space.</td></tr><tr><td>interface-name</td><td>This is an <i>in</i> parameter that is a bounded string containing the interface name. This must be the same as the value specified in the <i>idlmembername</i> and <i>idlmembernameX</i> copybooks (that is, of the form <i>IDL:name:version_number</i>). This string must be terminated by at least one space.</td></tr><tr><td>object-id</td><td>This is an <i>in</i> 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.</td></tr><tr><td>object-reference</td><td>This is an <i>out</i> parameter that contains the created object reference.</td></tr></table>	server-name	This is an <i>in</i> parameter that is a bounded string containing the server name. This must be the same as the value passed to ORBSRV. This string must be terminated by at least one space.	interface-name	This is an <i>in</i> parameter that is a bounded string containing the interface name. This must be the same as the value specified in the <i>idlmembername</i> and <i>idlmembernameX</i> copybooks (that is, of the form <i>IDL:name:version_number</i> ). This string must be terminated by at least one space.	object-id	This is an <i>in</i> 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 <i>out</i> parameter that contains the created object reference.
server-name	This is an <i>in</i> parameter that is a bounded string containing the server name. This must be the same as the value passed to ORBSRV. This string must be terminated by at least one space.								
interface-name	This is an <i>in</i> parameter that is a bounded string containing the interface name. This must be the same as the value specified in the <i>idlmembername</i> and <i>idlmembernameX</i> copybooks (that is, of the form <i>IDL:name:version_number</i> ). This string must be terminated by at least one space.								
object-id	This is an <i>in</i> 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 <i>out</i> parameter that contains the created object reference.								

**Example**

The example can be broken down as follows:

1. Consider the following IDL:

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

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

```
WORKING-STORAGE SECTION.

    01 WS-SERVER-NAME          PICTURE X(18) VALUE
                                "simple_persistent ".
    01 WS-SERVER-NAME-LEN      PICTURE 9(09) BINARY VALUE 17.

    01 WS-INTERFACE-NAME      PICTURE X(28) VALUE
                                "IDL:Simple/SimpleObject:1.0 ".
    01 WS-OBJECT-IDENTIFIER    PICTURE X(17) VALUE
                                "my_simple_object ".
    01 WS-SIMPLE-SIMPLEOBJECT  POINTER VALUE NULL.

PROCEDURE DIVISION.

    ...
    CALL "OBJNEW"      USING WS-SERVER-NAME
                                WS-INTERFACE-NAME
                                WS-OBJECT-IDENTIFIER
                                WS-SIMPLE-SIMPLEOBJECT.

    SET WS-OBJNEW TO TRUE.
    PERFORM CHECK-STATUS.
```

## Exceptions

---

A `CORBA::BAD_PARAM::INVALID_SERVER_NAME` exception is raised if the server name does not match the server name passed to `ORBSRV`.

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 `ORBSRV` is not called.

---

# OBJREL

Synopsis	<pre>OBJREL(inout POINTER object-reference) // Releases an object reference.</pre>		
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	<p>The parameter for OBJREL can be described as follows:</p> <table><tr><td>object-reference</td><td>This is an inout parameter that contains the valid object reference.</td></tr></table>	object-reference	This is an inout parameter that contains the valid object reference.
object-reference	This is an inout parameter that contains the valid object reference.		
Example	<p>The following is an example of how to use OBJREL in a client or server program:</p>		

```
WORKING-STORAGE SECTION.
01 WS-SIMPLE-SIMPLEOBJECT          POINTER VALUE NULL.
01 WS-SIMPLE-SIMPLEOBJECT-COPY     POINTER VALUE NULL.

PROCEDURE DIVISION.
...
* Note that the object reference will have been created, for
* example, by a call to OBJNEW.

        CALL "OBJDUP" USING WS-SIMPLE-SIMPLEOBJECT
                                WS-SIMPLE-SIMPLEOBJECT-COPY.
        SET WS-OBJDUP TO TRUE.
        PERFORM CHECK-STATUS.

        CALL "OBJREL" USING WS-SIMPLE-SIMPLEOBJECT-COPY.
        SET WS-OBJREL TO TRUE.
        PERFORM CHECK-STATUS.
```

**See also**

- 
- [“OBJDUP” on page 301.](#)
  - [“Object References and Memory Management” on page 393.](#)

# OBJRIR

Synopsis

```

OBJRIR(in X(nn) desired-service,
      out POINTER object-reference)
// Returns an object reference to an object through which a
// service such as the Naming Service can be used.

```

Usage

Common to clients and servers.

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_services()` in C++.

[Table 26](#) shows the common services available, along with the COBOL identifier assigned to each service. The COBOL identifiers are declared in the CORBA copybook.

**Table 26:** *Summary of Common Services and Their COBOL Identifiers*

Service	COBOL Identifier
InterfaceRepository	IFR-SERVICE
NameService	NAMING-SERVICE
TradingService	TRADING-SERVICE

Not all the services available in C++ are available in COBOL. Refer to the `list_initial_services` function in the *CORBA Programmer's Reference, C++* for details of all the available services.

Parameters

The parameters for `OBJRIR` can be described as follows:

<code>desired-service</code>	This is an <code>in</code> parameter that is a string specifying the desired service. This string is terminated by a space.
<code>object-reference</code>	This is an <code>out</code> parameter that contains an object reference for the desired service.



## Example

The example can be broken down as follows:

1. The following code is defined in the supplied CORBA copybook:

```
01 SERVICE-REQUESTED      PICTURE X(20)
                           VALUE SPACES.
    88 IFR-SERVICE        VALUE "InterfaceRepository ".
    88 NAMING-SERVICE      VALUE "NameService ".
    88 TRADING-SERVICE    VALUE "TradingService ".
```

2. The following is an example of how to use OBJRIR in a client or server program:

```
WORKING-STORAGE SECTION
    01 WS-NAMESERVICE-OBJ POINTER VALUE NULL.
PROCEDURE DIVISION.

    ...
    SET NAMING-SERVICE TO TRUE.
    CALL "OBJRIR"      USING SERVICE-REQUESTED
                           WS-NAMESERVICE-OBJ.

    SET WS-OBJRIR TO TRUE.
    PERFORM CHECK-STATUS.
```

## Exceptions

A CORBA::ORB::InvalidName exception is raised if the desired-service string is invalid.

---

# OBJTOSTR

**Synopsis**

---

```
OBJTOSTR(in POINTER object-reference,  
         out POINTER object-string)  
// Returns a stringified interoperable object reference (IOR)  
// from a valid object reference.
```

---

**Usage**

Common to clients and servers.

---

**Description**

The OBJTOSTR function returns a string representation of an object reference. It translates an object reference into a string, and the resulting value can then be stored or communicated in whatever ways strings are manipulated. A string representation of an object reference has an IOR: prefix followed by a series of hexadecimal octets. It is equivalent to calling CORBA::ORB::object\_to\_string() in C++.

Because an object reference is opaque and might also be used in other contexts, it is not recommended to use it in a context where it might be used in a different way. For example, it is not recommended to use it in a context where it might be used in a different way.

---

## Example

The following is an example of how to use OBJTOSTR in a client or server program:

```
WORKING-STORAGE SECTION.  
01 WS-SIMPLE-SIMPLEOBJECT          POINTER VALUE NULL.  
01 WS-IOR-PTR                     POINTER VALUE NULL.  
01 WS-IOR-STRING                   PICTURE X(2048) VALUE SPACES.  
01 WS-IOR-LEN                      PICTURE 9(09) BINARY VALUE 2048.  
  
PROCEDURE DIVISION.  
...  
* Note that the object reference will have been created, for  
* example, by a call to OBJNEW.  
  
    CALL "OBJTOSTR" USING WS-SIMPLE-SIMPLEOBJECT  
                          WS-IOR-PTR.  
    SET WS-OBJTOSTR TO TRUE.  
    PERFORM CHECK-STATUS.  
  
    CALL "STRGET" USING WS-IOR-PTR  
                      WS-IOR-LEN  
                      WS-IOR-STRING.  
    SET WS-STRGET TO TRUE.  
    PERFORM CHECK-STATUS.  
    DISPLAY "Interoperable object reference (IOR) equals "  
    WS-IOR-STRING.
```

## See also

["STRTOOBJ" on page 367.](#)

---

## ORBARGS

---

### Synopsis

```
ORBARGS(in X(nn) argument-string,
        in 9(09) BINARY argument-string-length,
        in X(nn) orb-name,
        in 9(09) BINARY 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, by making a call to `CORBA::ORB_init()` in C++. It first initializes an application in the ORB environment and then it returns the ORB pseudo-object reference to the application for use in future ORB calls.

Because applications do not initially have an object on which to invoke ORB calls, `ORB_init()` is a bootstrap call into the CORBA environment. Therefore, the `ORB_init()` call is part of the `CORBA` module but is not part of the `CORBA::ORB` class.

The `arg-list` is optional and is usually not set. The use of the `orb-name` is recommended, because if it is not specified, a default ORB name is used.

Special ORB identifiers (indicated by either the `orb-name` parameter or the `-ORBid` argument) are intended to uniquely identify each ORB used within the same location domain in a multi-ORB application. The ORB identifiers are allocated by the ORB administrator who is responsible for ensuring that the names are unambiguous.

When you are assigning ORB identifiers via `ORBARGS`, if the `orb-name` parameter has a value, any `-ORBid` arguments in the `argv` are ignored. However, all other ORB arguments in `argv` might be significant during the ORB initialization process. If the `orb-name` parameter is null, the ORB identifier is obtained from the `-ORBid` argument of `argv`. If the `orb-name` is null and there is no `-ORBid` argument in `argv`, the default ORB is returned in the call.

## Parameters

The parameters for `ORBARGS` can be described as follows:

<code>argument-string</code>	This is an <code>in</code> parameter that is a bounded string containing the argument list of the environment-specific data for the call. Refer to <a href="#">“ORB arguments”</a> for more details.
<code>argument-string-length</code>	This is an <code>in</code> parameter that specifies the length of the argument string list.
<code>orb-name</code>	This is an <code>in</code> 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.
<code>orb-name-length</code>	This is an <code>in</code> 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:

<code>-ORBboot_domain value</code>	This indicates where to get boot configuration information.
<code>-ORBdomain value</code>	This indicates where to get the ORB actual configuration information.
<code>-ORBid value</code>	This is the ORB identifier.
<code>-ORBname value</code>	This is the ORB name.

## Example

The following is an example of how to use ORBARGS in a client or server program:

```
WORKING-STORAGE SECTION.  
01 ARG-LIST                                PICTURE X(01) VALUE SPACES  
01 ARG-LIST-LEN                            PICTURE 9(09) BINARY VALUE 0.  
01 ORB-NAME                               PICTURE X(10) VALUE "simple_orb"  
01 ORB-NAME-LEN                           PICTURE 9(09) BINARY VALUE 10.  
  
PROCEDURE DIVISION.  
...  
    DISPLAY "Initializing the ORB".  
    CALL "ORBARGS" USING ARG-LIST  
                        ARG-LIST-LEN  
                        ORB-NAME  
                        ORB-NAME-LEN.  
    SET WS-ORBARGS TO TRUE.  
    PERFORM CHECK-STATUS.
```

## Exceptions

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

---

# ORBEXEC

## Synopsis

---

```
ORBEXEC(in POINTER object-reference,  
        in X(nn) operation-name,  
        inout buffer operation-buffer,  
        inout buffer user-exception-buffer)  
// Invokes an operation on the specified object.
```

---

## Usage

Client-specific. (Batch clients only.)

---

## Description

The `ORBEXEC` function allows a COBOL 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. `ORBEXEC` 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

---

operation-buffer	This is an <code>inout</code> parameter that contains a COBOL 01 level data item representing the data types that the operation supports.
user-exception-buffer	This is an <code>in</code> parameter that contains the COBOL representation of the user exceptions that the operation supports, as defined in the <i>idlmembername</i> copybook generated by the Orbix E2A IDL compiler. If the IDL operation supports no user exceptions, a dummy buffer is generated—this dummy buffer is not populated on the server side, and it is only used as the fourth (in this case, dummy) parameter to <code>ORBEXEC</code> .

### Example

The example can be broken down as follows:

1. Consider the following IDL:

```
// IDL
interface sample
{
    typedef string<10> Aboundedstring;
    exception MyException {Aboundedstring except_str; };
    Aboundedstring myoperation(in Aboundedstring instr,
        inout Aboundedstring inoutstr,
        out Aboundedstring outstr)
        raises(MyException);
};
```

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

#### Example 27: *The idlmembername Copybook (Sheet 1 of 3)*

```
*****
* Operation:      myoperation
* Mapped name:    myoperation
* Arguments:      <in> sample/Aboundedstring instr
*                 <inout> sample/Aboundedstring inoutstr
*                 <out> sample/Aboundedstring outstr
* Returns:        sample/Aboundedstring
* User Exceptions: sample/MyException
```



**Example 27:** *The idlmembername Copybook (Sheet 2 of 3)*

```

*****
* operation-buffer
01 SAMPLE-MYOPERATION-ARGS.
    03 INSTR                                PICTURE X(10).
    03 INOUTSTR                             PICTURE X(10).
    03 OUTSTR                               PICTURE X(10).
    03 RESULT                              PICTURE X(10).

*****
COPY EXAMPLX.
*****

*****
*
* Operation List section
* This lists the operations and attributes which an
* interface supports
*
*****

* The operation-name and its corresponding 88 level entry
01 SAMPLE-OPERATION                                PICTURE X(27).
    88 SAMPLE-MYOPERATION                          VALUE
        "myoperation:IDL:sample:1.0".
01 SAMPLE-OPERATION-LENGTH                        PICTURE 9(09)
                                                BINARY VALUE 27.
*****
*
* Typecode section
* This contains CDR encodings of necessary typecodes.
*
*****

01 EXAMPLE-TYPE                                PICTURE X(29).
    COPY CORBATYP.
    88 SAMPLE-ABOUNDEDSTRING                      VALUE
        "IDL:sample/Aboundedstring:1.0".
01 EXAMPLE-TYPE-LENGTH                        PICTURE S9(09)
                                                BINARY VALUE 29.

*****
* User exception block
*****
01 EX-SAMPLE-MYEXCEPTION                        PICTURE X(26)
                                                VALUE

```

**Example 27:** *The idlmembername Copybook (Sheet 3 of 3)*

```

"IDL:sample/MyException:1.0".
01 EX-SAMPLE-MYEXCEPTION-LENGTH          PICTURE 9(09)
                                           BINARY VALUE 26.

* user exception buffer
01 EXAMPLE-USER-EXCEPTIONS.
    03 EXCEPTION-ID                      POINTER
                                           VALUE NULL.
    03 D                                  PICTURE 9(10) BINARY
                                           VALUE 0.
        88 D-NO-USEREXCEPTION            VALUE 0.
        88 D-SAMPLE-MYEXCEPTION          VALUE 1.
    03 U                                  PICTURE X(10)
                                           VALUE LOW-VALUES.
    03 EXCEPTION-SAMPLE-MYEXCEPTION        REDEFINES U.
        05 EXCEPT-STR                  PICTURE X(10).

```

3. The following is an example of how to use ORBEXEC in a client program:

**Example 28:** *Using ORBEXEC in the Client Program (Sheet 1 of 2)*

```

WORKING-STORAGE SECTION.
    01 WS-SAMPLE-OBJ                      POINTER VALUE NULL.
    01 WS-EXCEPT-ID-STR                  PICTURE X(200) VALUES SPACES.

PROCEDURE DIVISION.
    ...
    *The SAMPLE-OBJ will have been created
    *with a previous call to api STRTOOBJ

        SET SAMPLE-MYOPERATION    TO TRUE
        DISPLAY "invoking Simple::" SAMPLE-OPERATION.
    * populate the in arguments
        MOVE "Hello " TO INSTR OF SAMPLE-MYOPERATION-ARGS.
    * populate the inout arguments

        MOVE "Server " TO INOUTSTR OF SAMPLE-MYOPERATION-ARGS.

        CALL "ORBEXEC"    USING WS-SAMPLE-OBJ
                                SAMPLE-OPERATION
                                SAMPLE-MYOPERATION-ARGS
                                SAMPLE-USER-EXCEPTIONS.

        SET WS-ORBEXEC TO TRUE.
        PERFORM CHECK-STATUS.
    * check if user exceptions thrown

```

**Example 28: Using ORBEXEC in the Client Program (Sheet 2 of 2)**

```

        EVALUATE TRUE
        WHEN D-NO-USEREXCEPTION
* no exception
* check inout arguments
        DISPLAY "In out parameter returned equals "
        INOUTSTR OF SAMPLE-MYOPERATION-ARGS
* check out arguments
        DISPLAY "Out parameter returned equals "
        OUTSTR OF SAMPLE-MYOPERATION-ARGS
* check return arguments
        DISPLAY "Return parameter returned equals "
        RESULT OF SAMPLE-MYOPERATION-ARGS
* MYEXCEPTION rasied by the server
        WHEN D-SAMPLE-MYEXCEPTION
            MOVE SPACES TO WS-EXCEPT-ID-STRING
*retrieve string value form the exception-id pointer
        CALL "STRGET" USING EXCEPTION-ID OF
            SAMPLE-USER-EXCEPTIONS
            EX-SAMPLE-MYEXCEPTION-LENGTH
            WS-EXCEPT-ID-STRING
        DISPLAY "Exception id equals "
        WS-EXCEPT-ID-STRING

*Check the values of the returned exception which
*in this example is a bounded string
        DISPLAY "Exception value retuned "
        EXCEPT-STR OF EXAMPLE-USER-EXCEPTIONS
        CALL "STRFREE" EXCEPTION-ID OF SAMPLE-USER-EXCEPTIONS
        SET WS-STRFREE TO TRUE
        PERFORM CHECK-STATUS
* Initialize for the next ORBEXEC call
        SET D-NO-USEREXCEPTION TO TRUE
        END-EVALUATE.

```

## Exceptions

---

A `CORBA::BAD_INV_ORDER::INTERFACE_NOT_REGISTERED` exception is raised if the client tries to invoke an operation on an interface that has not been registered via `ORBREG`.

A `CORBA::BAD_PARAM::INVALID_DISCRIMINATOR_TYPECODE` exception is raised if the discriminator typecode is invalid when marshalling a union type.

A `CORBA::BAD_PARAM::UNKNOWN_OPERATION` exception is raised if the operation is not valid for the interface.

A `CORBA::BAD_PARAM::UNKNOWN_TYPECODE` exception is raised if the typecode cannot be determined when marshalling an `any` type or a user exception.

A `CORBA::DATA_CONVERSION::VALUE_OUT_OF_RANGE` exception is raised if the value is determined to be out of range when marshalling a long, short, unsigned short, unsigned long, long long, or unsigned long long type.

---

## See also

- [“COAGET” on page 281](#).
- [“COAPUT” on page 286](#).
- The `BANK` demonstration in `orbixhlq.DEMOS.COBOL.SRC` for a complete example of how to use `ORBEXEC`.

# ORBHOST

Synopsis	<pre>ORBHOST(in 9(09) BINARY hostname-length,         out X(nn) hostname) // Returns the hostname of the server</pre>				
Usage	Server-specific.				
Description	<p>The ORBHOST function returns the hostname of the machine on which the server is running.</p> <p><b>Note:</b> This is only applicable if TCP/IP is being used on the host machine.</p>				
Parameters	<p>The parameters for ORBEXEC can be described as follows:</p> <table><tr><td>hostname-length</td><td>This is an <code>in</code> parameter that specifies the length of the hostname.</td></tr><tr><td>hostname</td><td>This is an <code>out</code> parameter that is a bounded string used to retrieve the hostname.</td></tr></table>	hostname-length	This is an <code>in</code> parameter that specifies the length of the hostname.	hostname	This is an <code>out</code> parameter that is a bounded string used to retrieve the hostname.
hostname-length	This is an <code>in</code> parameter that specifies the length of the hostname.				
hostname	This is an <code>out</code> parameter that is a bounded string used to retrieve the hostname.				
Example	<p>The following is an example of how to use ORBHOST in a server program:</p> <pre>WORKING-STORAGE SECTION. 01 HOST-NAME                PICTURE X(255). 01 HOST-NAME-LEN            PICTURE 9(09) BINARY                              VALUE 255.  PROCEDURE DIVISION. ...     CALL "ORBHOST" USING HOST-NAME-LENGTH                         HOST-NAME.     SET WS-ORBHOST TO TRUE.     PERFORM CHECK-STATUS.     DISPLAY "Hostname equals " HOST-NAME</pre>				

## Exceptions

---

A `CORBA::BAD_PARAM::LENGTH_TOO_SMALL` exception is raised if the length of the string containing the hostname is greater than the `hostname-length` parameter.

# ORBREG

Synopsis	<div> <div>ORBREG(in <i>buffer</i> interface-description)</div> <div>// Describes an IDL interface to the COBOL runtime.</div> </div>
Usage	<div>Common to clients and servers.</div>
Description	<div> <p>The ORBREG function registers an interface with the COBOL runtime, by using the interface description that is stored in the <i>idlmembernameX</i> copybook generated by the Orbix E2A IDL compiler. Each interface within the IDL member has a 01 level, which is the parameter to be passed to the ORBREG call.</p> <p>The Orbix 2000 IDL compiler generates a 01 level in the <i>idlmembernameX</i> copybook for each interface in the IDL member. Each 01 level that is generated fully describes the interface to the COBOL runtime; for example, the interface name, what it inherits from, each operation, its parameters and user exceptions, and all the associated typecodes. The <i>idlmembernameX</i> copybook cannot be amended by the user, because doing so can cause unpredictable results at runtime.</p> <p>You must call ORBREG for every interface that the client or server uses. However, it is to be called only once for each interface; therefore, you should place the calls in the client and server mainline programs.</p> </div>
Parameters	<div> <div>The parameter for ORBREG can be described as follows:</div> <div> <div>interface-description</div> <div>This is an in parameter that contains the address of the interface definition, which is defined as a 01 level in the <i>idlmembernameX</i> copybook.</div> </div> </div>

**Example**

The example can be broken down as follows:

1. Consider the following IDL:

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

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the *idlmembername*x copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

```
01 SIMPLE-SIMPLEOBJECT-INTERFACE.
   03 FILLER PIC X(160) VALUE X"0000005C00000058C9C4D37
-      "AE2899497938561E28994979385D682918583A37AF14BF
-      "000000000040000000EC9C4D37AE2899497938561E2899
-      "4979385D682918583A37AF14BF0000000001E289949793
-      "85D682918583A300FFFFFF00000004C9C4D37AE2899497
-      "938561E28994979385D682918583A37AF14BF000000000
-      "180000000000000001838193936D94850000000000000
-      "000000000000000000".
```

3. The following is an example of how to use ORBREG in a client or server program:

```
WORKING-STORAGE SECTION.

    COPY SIMPLE.

PROCEDURE DIVISION.
* Register interface(s) after ORB initialization
    DISPLAY "Registering the Interface".

    CALL "ORBREG" USING
    SIMPLE-SIMPLEOBJECT-INTERFACE.
    SET WS-ORBREG TO TRUE.
    PERFORM CHECK-STATUS
```



## 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.

---

# ORBSRV

Synopsis	<pre>ORBSRV(in X(nn) server-name,       in 9(09) BINARY server-name-length) // Sets the server name for the current server process.</pre>				
Usage	Server-specific.				
Description	The ORBSRV function sets the server name for the current server. This should be contained in the server mainline program, and should be called only once, after calling ORBARGS.				
Parameters	<p>The parameters for ORBSRV can be described as follows:</p> <table><tr><td>server-name</td><td>This is an in parameter that is a bounded string containing the server name.</td></tr><tr><td>server-name-length</td><td>This is an in parameter that specifies the length of the string containing the server name.</td></tr></table>	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.
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	<p>The following is an example of how to use ORBSRV in a server program:</p> <pre>WORKING-STORAGE SECTION. 01 SERVER-NAME          PICTURE X(17) VALUE "simple_persistent". 01 SERVER-NAME-LEN      PICTURE 9(09) BINARY VALUE 17. ... PROCEDURE DIVISION. ... * After ORBARGS call.   CALL "ORBSRV" USING SERVER-NAME                       SERVER-NAME-LEN.   SET WS-ORBSRV TO TRUE.   PERFORM CHECK-STATUS.</pre>				
Exceptions	A CORBA::BAD_INV_ORDER::SERVER_NAME_ALREADY_SET exception is raised if ORBSRV is called more than once.				

# ORBSTAT

Synopsis	<pre>ORBSTAT(<i>in buffer</i> status-buffer) // Registers the status information block.</pre>		
Usage	Common to both clients and servers.		
Description	<p>The ORBSTAT function registers the supplied status information block to the COBOL runtime. The status of any COBOL runtime call can then be checked, for example, to test if a call has completed successfully.</p> <p>The ORBIX-STATUS-INFORMATION structure is defined in the supplied CORBA copybook. A CHKERRS (for batch) or CERRSMFA (for IMS or CICS) copybook is also provided, which contains a CHECK-STATUS function that can be called after each API call, to check if a system exception has occurred. Alternatively, this can be modified or replaced for the system environment.</p> <p>You should call ORBSTAT once, as the first API call, in your server mainline and client programs. If it is not called, and an exception occurs at runtime, the application terminates with the following message:</p> <div> <p>An exception has occurred but ORBSTAT has not been called. Place the ORBSTAT API call in your application, compile and rerun. Exiting now.</p> </div>		
Parameters	<p>The parameters for ORBSTAT can be described as follows:</p> <table> <tr> <td>status-buffer</td><td>This is an <i>in</i> parameter that contains a COBOL 01 level data item representing the status information block defined in the CORBA copybook. This buffer is populated when a CORBA system exception occurs during subsequent API calls. Refer to <a href="#">“Definition of status information block”</a> for more details of how it is defined.</td></tr> </table>	status-buffer	This is an <i>in</i> parameter that contains a COBOL 01 level data item representing the status information block defined in the CORBA copybook. This buffer is populated when a CORBA system exception occurs during subsequent API calls. Refer to <a href="#">“Definition of status information block”</a> for more details of how it is defined.
status-buffer	This is an <i>in</i> parameter that contains a COBOL 01 level data item representing the status information block defined in the CORBA copybook. This buffer is populated when a CORBA system exception occurs during subsequent API calls. Refer to <a href="#">“Definition of status information block”</a> for more details of how it is defined.		

# Definition of status information block

ORBIX-STATUS-INFORMATION is defined in the CORBA copybook as follows:

## Example 29: ORBIX-STATUS-INFORMATION Definition (Sheet 1 of 2)

```

*
** This data item must be originally set by calling the
** ORBSTAT api.
** This data item is then used to determine the status of
** each api called (eg COAGET, ORBEXEC).
**
** If the call was successful then CORBA-EXCEPTION and
** CORBA-MINOR-CODE will be both set to 0 and
** COMPLETION-STATUS-YES will be set to true.
**
** EXCEPTION-TEXT is a pointer to the text of the exception.
** STRGET must be used to extract this text.
** (Refer to CHKERRS or CERRSMFA Copybooks for more details).
*
01 ORBIX-STATUS-INFORMATION IS EXTERNAL.
   03 CORBA-EXCEPTION                                PICTURE 9(5) BINARY.
      88 CORBA-NO-EXCEPTION                            VALUE 0.
      88 CORBA-UNKNOWN                                VALUE 1.
      88 CORBA-BAD-PARAM                              VALUE 2.
      88 CORBA-NO-MEMORY                              VALUE 3.
      88 CORBA-IMP-LIMIT                              VALUE 4.
      88 CORBA-COMM-FAILURE                            VALUE 5.
      88 CORBA-INV-OBJREF                              VALUE 6.
      88 CORBA-NO-PERMISSION                          VALUE 7.
      88 CORBA-INTERNAL                                VALUE 8.
      88 CORBA-MARSHAL                                VALUE 9.
      88 CORBA-INITIALIZE                              VALUE 10.
      88 CORBA-NO-IMPLEMENT                            VALUE 11.
      88 CORBA-BAD-TYPECODE                            VALUE 12.
      88 CORBA-BAD-OPERATION                          VALUE 13.
      88 CORBA-NO-RESOURCES                            VALUE 14.
      88 CORBA-NO-RESPONSE                            VALUE 15.
      88 CORBA-PERSIST-STORE                          VALUE 16.
      88 CORBA-BAD-INV-ORDER                          VALUE 17.
      88 CORBA-TRANSIENT                              VALUE 18.
      88 CORBA-FREE-MEM                                VALUE 19.
      88 CORBA-INV-IDENT                              VALUE 20.
      88 CORBA-INV-FLAG                                VALUE 21.
      88 CORBA-INTF-REPOS                              VALUE 22.
      88 CORBA-BAD-CONTEXT                            VALUE 23.
      88 CORBA-OBJ-ADAPTER                            VALUE 24.

```

**Example 29: ORBIX-STATUS-INFORMATION Definition (Sheet 2 of 2)**

88	CORBA-DATA-CONVERSION	VALUE 25.
88	CORBA-OBJECT-NOT-EXIST	VALUE 26.
88	CORBA-TRANSACTION-REQUIRED	VALUE 27.
88	CORBA-TRANSACTION-ROLLEDBACK	VALUE 28.
88	CORBA-INVALID-TRANSACTION	VALUE 29.
88	CORBA-INV-POLICY	VALUE 30.
88	CORBA-REBIND	VALUE 31.
88	CORBA-TIMEOUT	VALUE 32.
88	CORBA-TRANSACTION-UNAVAILABLE	VALUE 33.
88	CORBA-TRANSACTION-MODE	VALUE 34.
88	CORBA-BAD-QOS	VALUE 35.
88	CORBA-CODESET-INCOMPATIBLE	VALUE 36.
03	COMPLETION-STATUS	PICTURE 9(5) BINARY
88	COMPLETION-STATUS-YES	VALUE 0.
88	COMPLETION-STATUS-NO	VALUE 1.
88	COMPLETION-STATUS-MAYBE	VALUE 2.
03	EXCEPTION-MINOR-CODE	PICTURE S9(10) BINARY
03	EXCEPTION-NUMBER REDEFINES EXCEPTION-MINOR-CODE	PICTURE S9(10) BINARY.
03	EXCEPTION-TEXT	USAGE IS POINTER

## Example

The following is an example of how to use ORBSTAT in a server mainline or client program:

```
WORKING-STORAGE SECTION.  
    COPY CORBA  
...  
PROCEDURE DIVISION.  
  
    CALL "ORBSTAT" USING ORBIX-STATUS-INFORMATION.  
  
    DISPLAY "Initializing the ORB".  
  
    CALL "ORBARGS" USING ARG-LIST  
                        ARG-LEN  
                        ORB-NAME  
                        ORB-NAME-LEN.  
    SET WS-ORBARGS TO TRUE.  
    PERFORM CHECK-STATUS.  
...  
EXIT-PRG.  
    STOP RUN.  
...  
COPY CHKERRS.
```

**Note:** The COPY CHKERRS statement in the preceding example is used in batch programs. It is replaced with COPY CERRSMFA in IMS or CICS server programs.

## Exceptions

A CORBA::BAD\_INV\_ORDER::STAT\_ALREADY\_CALLED exception is raised if ORBSTAT is called more than once with a different ORBIX-STATUS-INFORMATION block.

---

# ORBTIME

Synopsis	<pre>ORBTIME(in 9(04) BINARY timeout-type         in 9(09) BINARY timeout-value) // Used by clients for setting the call timeout. // Used by servers for setting the event timeout.</pre>				
Usage	Common to clients and servers. (Batch only.)				
Description	<p>The ORBTIME function provides:</p> <ul style="list-style-type: none"><li>• 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. The value only comes into effect after the connection has been established.</li><li>• Event timeout support to servers. This means that it specifies how long a server should wait between connection requests.</li></ul>				
Parameters	<p>The parameters for ORBTIME can be described as follows:</p> <table><tr><td>timeout-type</td><td>This is an <code>in</code> parameter that determines whether call timeout or event timeout functionality is required. It must be set to one of the two values defined in the CORBA copybook for the ORBIX-TIMEOUT-TYPE. In this case, value 1 corresponds to event timeout, and value 2 corresponds to call timeout.</td></tr><tr><td>timeout-value</td><td>This is an <code>in</code> parameter that specifies the timeout value in milliseconds.</td></tr></table>	timeout-type	This is an <code>in</code> parameter that determines whether call timeout or event timeout functionality is required. It must be set to one of the two values defined in the CORBA copybook for the ORBIX-TIMEOUT-TYPE. In this case, value 1 corresponds to event timeout, and value 2 corresponds to call timeout.	timeout-value	This is an <code>in</code> parameter that specifies the timeout value in milliseconds.
timeout-type	This is an <code>in</code> parameter that determines whether call timeout or event timeout functionality is required. It must be set to one of the two values defined in the CORBA copybook for the ORBIX-TIMEOUT-TYPE. In this case, value 1 corresponds to event timeout, and value 2 corresponds to call timeout.				
timeout-value	This is an <code>in</code> parameter that specifies the timeout value in milliseconds.				

**Server example**

On the server side, `ORBTIME` must be called immediately before calling `COARUN`. After `COARUN` has been called, the event timeout value cannot be changed. For example:

```
...
01 WS-TIMEOUT-VALUE          PICTURE 9(09) BINARY VALUE 0.
...
PROCEDURE DIVISION.
...
*set the timeout value to two minutes
MOVE 120000 TO WS-TIMEOUT-VALUE
SET EVENT-TIMEOUT TO TRUE.
CALL "ORBTIME" USING ORBIX-TIMEOUT-TYPE
                    WS-TIMEOUT-VALUE.
SET WS-ORBTIME TO TRUE.
PERFORM CHECK-STATUS.
CALL "COARUN" .
...
```

**Client example**

On the client side, `ORBTIME` must be called before calling `ORBEXEC`. For example:

```
...
*set the timeout value to two minutes
MOVE 120000 TO WS-TIMEOUT-VALUE
SET CALL-TIMEOUT TO TRUE.
CALL "ORBTIME" USING ORBIX-TIMEOUT-TYPE
                    WS-TIMEOUT-VALUE.
SET WS-ORBTIME TO TRUE.
PERFORM CHECK-STATUS.
CALL "ORBEXEC" ...
```

**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 `ORBIX-TIMEOUT-TYPE` in the `CORBA` copybook.



---

# SEQALLOC

Synopsis	<pre>SEQALLOC(in 9(09) BINARY sequence-size,          in X(nn) typecode-key,          in 9(09) BINARY typecode-key-length,          inout <i>buffer</i> sequence-control-data) // Allocates memory for an unbounded sequence</pre>								
Usage	Common to clients and servers.								
Description	<p>The <code>SEQALLOC</code> function allocates initial storage for an unbounded sequence. You must call <code>SEQALLOC</code> before you call <code>SEQSET</code> for the first time. The length supplied to the function is the initial sequence size requested. The typecode supplied to <code>SEQALLOC</code> must be the sequence typecode.</p> <p><b>Note:</b> You can use <code>SEQALLOC</code> only on unbounded sequences.</p>								
Parameters	<p>The parameters for <code>SEQALLOC</code> can be described as follows:</p> <table><tr><td><code>sequence-size</code></td><td>This is an <code>in</code> parameter that specifies the maximum expected size of the sequence.</td></tr><tr><td><code>typecode-key</code></td><td>This is an <code>in</code> parameter that contains a 01 level data item representing the typecode key, as defined in the <i>idlmembername</i> copybook generated by the Orbix E2A IDL compiler. This is a bounded string.</td></tr><tr><td><code>typecode-key-length</code></td><td>This is an <code>in</code> parameter that specifies the length of the typecode key, as defined in the <i>idlmembername</i> copybook generated by the Orbix E2A IDL compiler.</td></tr><tr><td><code>sequence-control-data</code></td><td>This is an <code>inout</code> parameter that contains the unbounded sequence control data.</td></tr></table> <p><b>Note:</b> The typecode keys are defined as level 88 data items in the <i>idlmembername</i> copybook generated by the Orbix E2A IDL compiler.</p>	<code>sequence-size</code>	This is an <code>in</code> parameter that specifies the maximum expected size of the sequence.	<code>typecode-key</code>	This is an <code>in</code> parameter that contains a 01 level data item representing the typecode key, as defined in the <i>idlmembername</i> copybook generated by the Orbix E2A IDL compiler. This is a bounded string.	<code>typecode-key-length</code>	This is an <code>in</code> parameter that specifies the length of the typecode key, as defined in the <i>idlmembername</i> copybook generated by the Orbix E2A IDL compiler.	<code>sequence-control-data</code>	This is an <code>inout</code> parameter that contains the unbounded sequence control data.
<code>sequence-size</code>	This is an <code>in</code> parameter that specifies the maximum expected size of the sequence.								
<code>typecode-key</code>	This is an <code>in</code> parameter that contains a 01 level data item representing the typecode key, as defined in the <i>idlmembername</i> copybook generated by the Orbix E2A IDL compiler. This is a bounded string.								
<code>typecode-key-length</code>	This is an <code>in</code> parameter that specifies the length of the typecode key, as defined in the <i>idlmembername</i> copybook generated by the Orbix E2A IDL compiler.								
<code>sequence-control-data</code>	This is an <code>inout</code> parameter that contains the unbounded sequence control data.								

**Example**

The example can be broken down as follows:

1. Consider the following IDL:

```
// IDL
interface example
{
    typedef sequence<long> unboundedseq;
    unboundedseq myop();
};
```

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

**Example 30:** *The idlmembername Copybook (Sheet 1 of 2)*

```
*****
* Operation:          myop
* Mapped name:        myop
* Arguments:          None
* Returns:            example/unboundedseq
* User Exceptions:    none
*****
01 EXAMPLE-MYOP-ARGS.
   03 RESULT-1.
       05 RESULT                                PICTURE S9(10) BINARY.
   03 RESULT-SEQUENCE.
       05 SEQUENCE-MAXIMUM                      PICTURE 9(09) BINARY
                                              VALUE 0.
       05 SEQUENCE-LENGTH                      PICTURE 9(09) BINARY
                                              VALUE 0.
       05 SEQUENCE-BUFFER                      POINTER
                                              VALUE NULL.
       05 SEQUENCE-TYPE                        POINTER
                                              VALUE NULL.
*****
*
* Operation List section
* This lists the operations and attributes which an
* interface supports
*
*****
01 EXAMPLE-OPERATION                                PICTURE X(21).
```

**Example 30: The idlmembername Copybook (Sheet 2 of 2)**

```

      88 EXAMPLE-MYOP                                VALUE
      "myop:IDL:example:1.0".
01 EXAMPLE-OPERATION-LENGTH                          PICTURE 9(09) BINARY
                                                    VALUE 21.
*****
*
* Typecode section
* This contains CDR encodings of necessary typecodes.
*
*****
01 EXAMPLE-TYPE                                      PICTURE X(28).
      COPY CORBATYP.
      88 EXAMPLE-UNBOUNDEDSEQ                        VALUE
      "IDL:example/unboundedseq:1.0".
      88 EXAMPLE                                    VALUE
      "IDL:example:1.0".
01 EXAMPLE-TYPE-LENGTH                              PICTURE S9(09)
                                                    BINARY VALUE 28.

```

- The following is an example of how to use SEQALLOC in a client or server program:

**Example 31: Using SEQALLOC in Client or Server (Sheet 1 of 2)**

```

WORKING-STORAGE SECTION.
01 WS-MAX-ELEMENTS                                PICTURE 9(09) BINARY
                                                    VALUE 10.
01 WS-CURRENT-ELEMENT                            PICTURE 9(09) BINARY
                                                    VALUE 0.

      DO-EXAMPLE-MYOP.
      CALL "COAGET" USING EXAMPLE-MYOP-ARGS.
      SET WS-COAGET TO TRUE.
      PERFORM CHECK-STATUS.
      * initialize the maximum and length fields.

      *   MOVE WS-MAX-ELEMENTS TO SEQUENCE-MAXIMUM OF
      MOVE 0                                TO SEQUENCE-MAXIMUM OF
                                                    EXAMPLE-MYOP-ARGS.

      MOVE 0                                TO SEQUENCE-LENGTH OF
                                                    EXAMPLE-MYOP-ARGS.

      * Initialize the sequence element data
      MOVE 0 TO RESULT OF
      RESULT-1 OF

```

**Example 31:** *Using SEQALLOC in Client or Server (Sheet 2 of 2)*

```

EXAMPLE-MYOP-ARGS.
* set the typecode of the sequence
  SET EXAMPLE-UNBOUNDEDSEQ TO TRUE.
* Allocate memory for the unbounded sequence.
* NOTE: SEQUENCE-MAXIMUM is set to WS-MAX-ELEMENTS after
*      SEQALLOC call
  CALL "SEQALLOC" USING WS-MAX-ELEMENTS
                        EXAMPLE-TYPE
                        EXAMPLE-TYPE-LENGTH
                        RESULT-SEQUENCE OF
                        EXAMPLE-MYOP-ARGS.

  SET WS-SEQALLOC TO TRUE.
  PERFORM CHECK-STATUS.
* Now ready to populate the sequence see SEQSET
*****
* Check Errors Copybook
*****
COPY CHKERRS.

```

**Note:** The COPY CHKERRS statement in the preceding example is used in batch programs. It is replaced with COPY CERRSMFA in IMS or CICS server programs.

**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 344.](#)
- [“Unbounded Sequences and Memory Management” on page 385.](#)

# SEQDUP

## Synopsis

```
SEQDUP(in buffer sequence-control-data,  
       out buffer dupl-seq-control-data)  
// Duplicates an unbounded sequence control block.
```

## Usage

Common to clients and servers.

## Description

The `SEQDUP` 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, you must call `SEQFREE` to free the memory allocated to it.

You can call `SEQDUP` only on unbounded sequences.

## Parameters

The parameters for `SEQDUP` can be described as follows:

`sequence-control-data` This is an `in` parameter that contains the unbounded sequence control data.

`dupl-seq-control-data` This is an `out` parameter that contains the duplicated unbounded sequence control data block.

## Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface example  
{  
    typedef sequence<long> unboundedseq;  
    unboundedseq myop();  
};
```

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

**Example 32:** *The idlmembername Copybook (Sheet 1 of 2)*

```
*****
* Operation:      myop
* Mapped name:    myop
* Arguments:      None
* Returns:        example/unboundedseq
* User Exceptions: none
*****
01 EXAMPLE-MYOP-ARGS.
  03 RESULT-1.
    05 RESULT                                PICTURE S9(10) BINARY.
  03 RESULT-SEQUENCE.
    05 SEQUENCE-MAXIMUM                      PICTURE 9(09) BINARY
                                              VALUE 0.
    05 SEQUENCE-LENGTH                      PICTURE 9(09) BINARY
                                              VALUE 0.
    05 SEQUENCE-BUFFER                      POINTER
                                              VALUE NULL.
    05 SEQUENCE-TYPE                        POINTER
                                              VALUE NULL.
*****
*
* Operation List section
* This lists the operations and attributes which an
* interface supports
*
*****

01 EXAMPLE-OPERATION                      PICTURE X(21).
  88 EXAMPLE-MYOP                        VALUE
                                          "myop:IDL:example:1.0".
01 EXAMPLE-OPERATION-LENGTH              PICTURE 9(09) BINARY
                                          VALUE 21.
*****
*
* Typecode section
* This contains CDR encodings of necessary typecodes.
*
*****
```

**Example 32:** *The idlmembername Copybook (Sheet 2 of 2)*

```

01 EXAMPLE-TYPE                                PICTURE X(28).
      COPY CORBATYP.
      88 EXAMPLE-UNBOUNDEDSEQ                  VALUE
        "IDL:example/unboundedseq:1.0".
      88 EXAMPLE                              VALUE
        "IDL:example:1.0".
01 EXAMPLE-TYPE-LENGTH                        PICTURE S9(09) BINARY
      VALUE 28.

```

3. The following is an example of how to use SEQDUP in a client or server program:

**Example 33:** *Using SEQDUP in Client or Server (Sheet 1 of 2)*

```

WORKING-STORAGE SECTION.
01 WS-CURRENT-ELEMENT                        PICTURE 9(09) BINARY
      VALUE 0.
01 WS-ARGS.
      03 COPIED-1.
        05 COPIED-VALUE                      PICTURE S9(10) BINARY.
      03 COPIED-SEQUENCE.
        05 SEQUENCE-MAXIMUM                  PICTURE 9(09) BINARY
          VALUE 0.
        05 SEQUENCE-LENGTH                  PICTURE 9(09) BINARY
          VALUE 0.
        05 SEQUENCE-BUFFER                  POINTER
          VALUE NULL.
        05 SEQUENCE-TYPE                    POINTER
          VALUE NULL.

PROCEDURE DIVISION.

      CALL "ORBEXEC"      USING EXAMPLE-OBJ
                              EXAMPLE-OPERATION
                              EXAMPLE-MYOP-ARGS
                              EXAMPLE-USER-EXCEPTIONS.

      SET WS-ORBEXEC TO TRUE.
      PERFORM CHECK-STATUS.
      * Make a copy of the unbounded sequence
      CALL "SEQDUP" USING RESULT-SEQUENCE OF
                              EXAMPLE-MYOP-ARGS
                              COPIED-SEQUENCE OF
                              WS-ARGS.
      SET WS-SEQDUP TO TRUE.

```

**Example 33: Using SEQDUP in Client or Server (Sheet 2 of 2)**

```

PERFORM CHECK-STATUS.

* Release the memory allocated by SEQALLOC
* Refer to memory management chapter on when to call this
* api. * NOTE: The SEQUENCE-MAXIMUM and SEQUENCE-LENGTH
* are not initialized.

CALL "SEQFREE" USING RESULT-SEQUENCE OF
                                EXAMPLE-MYOP-ARGS.
SET WS-SEQFREE TO TRUE.
PERFORM CHECK-STATUS.

* Get each of the 10 elements in the copied sequence.
PERFORM VARYING WS-CURRENT-ELEMENT
    FROM 1 BY 1 UNTIL
        WS-CURRENT-ELEMENT >
            SEQUENCE-LENGTH OF
                WS-ARGS
* Get the current element in the copied sequence
CALL "SEQGET" USING COPIED-SEQUENCE OF
                    WS-ARGS
                    WS-CURRENT-ELEMENT
                    COPIED-VALUE OF
                    COPIED-1 OF
                    WS-ARGS

SET WS-SEQGET TO TRUE
PERFORM CHECK-STATUS
DISPLAY "Element data value equals "
        COPIED-VALUE OF
        COPIED-1 OF
        WS-ARGS

END-PERFORM.

EXIT-PRG.
=====
STOP RUN.
*****
* Check Errors Copybook
*****
COPY CHKERRS.

```



**Note:** The `COPY CHKERRS` statement in the preceding example is used in batch programs. It is replaced with `COPY CERRSMFA` in IMS or CICS server programs.

---

**Exceptions**

A `CORBA::BAD_PARAM::INVALID_SEQUENCE` exception is raised if the sequence has not been set up correctly.

---

**See also**

- [“SEQFREE” on page 344.](#)
- [“Unbounded Sequences and Memory Management” on page 385.](#)

---

# SEQFREE

Synopsis	<pre>SEQFREE(inout <i>buffer</i> sequence-control-data) // Frees the memory allocated to an unbounded sequence.</pre>
Usage	Common to clients and servers.
Description	<p>The <code>SEQFREE</code> function releases storage assigned to an unbounded sequence. (Storage is assigned to a sequence by calling <code>SEQALLOC</code>.) Do not try to use the sequence again after freeing its memory, because doing so might result in a runtime error.</p> <p>You can use <code>SEQFREE</code> only on unbounded sequences. Refer to the <a href="#">“Memory Handling” on page 383</a> for details of when it should be called.</p>
Parameters	<p>The parameter for <code>SEQFREE</code> can be described as follows:</p> <p><code>sequence-control-data</code> This is an <code>inout</code> parameter that contains the unbounded sequence control data.</p>
Example	<p>The example can be broken down as follows:</p> <ol style="list-style-type: none"><li>1. Consider the following IDL:</li></ol>

```
// IDL
interface example
{
    typedef sequence<long> unboundedseq;
    unboundedseq myop();
};
```

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

**Example 34:** *The idlmembername Copybook (Sheet 1 of 2)*

```
*****
* Operation:      myop
* Mapped name:    myop
* Arguments:      None
* Returns:        example/unboundedseq
* User Exceptions: none
*****
01 EXAMPLE-MYOP-ARGS.
   03 RESULT-1.
       05 RESULT                                PICTURE S9(10) BINARY.
   03 RESULT-SEQUENCE.
       05 SEQUENCE-MAXIMUM                      PICTURE 9(09) BINARY
                                                VALUE 0.
       05 SEQUENCE-LENGTH                      PICTURE 9(09) BINARY
                                                VALUE 0.
       05 SEQUENCE-BUFFER                      POINTER
                                                VALUE NULL.
       05 SEQUENCE-TYPE                        POINTER
                                                VALUE NULL.
*****
*
* Operation List section
* This lists the operations and attributes which an
* interface supports
*
*****
01 EXAMPLE-OPERATION                                PICTURE X(21).
   88 EXAMPLE-MYOP                                VALUE
       "myop:IDL:example:1.0".
01 EXAMPLE-OPERATION-LENGTH                      PICTURE 9(09) BINARY
                                                VALUE 21.
*****
*
* Typecode section
* This contains CDR encodings of necessary typecodes.
*
*****
01 EXAMPLE-TYPE                                PICTURE X(28).
```

**Example 34:** *The idlmembername Copybook (Sheet 2 of 2)*

```
COPY CORBATYP.
88 EXAMPLE-UNBOUNDEDESEQ          VALUE
   "IDL:example/unboundedseq:1.0".
88 EXAMPLE                        VALUE
   "IDL:example:1.0".
01 EXAMPLE-TYPE-LENGTH            PICTURE S9(09)
                                   BINARY VALUE 28.
```

3. The following is an example of how to use SEQFREE in a client or server program:

```
WORKING-STORAGE SECTION.
01 WS-MAX-ELEMENTS                PICTURE 9(09) BINARY
                                   VALUE 10.
01 WS-CURRENT-ELEMENT             PICTURE 9(09) BINARY
                                   VALUE 0.

* Release the memory allocated by SEQUALLOC
* Refer to memory management chapter on when to call this
* api.
* NOTE: The SEQUENCE-MAXIMUM and SEQUENCE-LENGTH are
*       not initialized.
      CALL "SEQFREE" USING RESULT-SEQUENCE OF
                                   EXAMPLE-MYOP-ARGS.
      SET WS-SEQFREE TO TRUE.
      PERFORM CHECK-STATUS.

*****
* Check Errors Copybook
*****
COPY CHKERS.
```

**Note:** The COPY CHKERS statement in the preceding example is used in batch programs. It is replaced with COPY CERRSMFA in IMS or CICS server programs.

See also

[“Unbounded Sequences and Memory Management” on page 385.](#)

---

# SEQGET

Synopsis	<pre>SEQGET(in sequence sequence-control-data,        in 9(09) BINARY element-number,        out buffer sequence-data) // Retrieves the specified element from an unbounded sequence.</pre>						
Usage	Common to clients and servers.						
Description	<p>The <code>SEQGET</code> function provides access to a specific element of an unbounded sequence. The data is copied from the specified element into the supplied data area (that is, into the <code>sequence-data</code> parameter).</p> <p>You can use <code>SEQGET</code> only on unbounded sequences.</p>						
Parameters	<p>The parameter for <code>SEQGET</code> can be described as follows:</p> <table><tr><td><code>sequence-control-data</code></td><td>This is an <code>in</code> parameter that contains the unbounded sequence control data.</td></tr><tr><td><code>element-number</code></td><td>This is an <code>in</code> parameter that specifies the index of the element number to be retrieved.</td></tr><tr><td><code>sequence-data</code></td><td>This is an <code>out</code> parameter that contains the buffer to which the sequence data is to be copied.</td></tr></table>	<code>sequence-control-data</code>	This is an <code>in</code> parameter that contains the unbounded sequence control data.	<code>element-number</code>	This is an <code>in</code> parameter that specifies the index of the element number to be retrieved.	<code>sequence-data</code>	This is an <code>out</code> parameter that contains the buffer to which the sequence data is to be copied.
<code>sequence-control-data</code>	This is an <code>in</code> parameter that contains the unbounded sequence control data.						
<code>element-number</code>	This is an <code>in</code> parameter that specifies the index of the element number to be retrieved.						
<code>sequence-data</code>	This is an <code>out</code> parameter that contains the buffer to which the sequence data is to be copied.						

Example	<p>The example can be broken down as follows:</p> <ol style="list-style-type: none"><li>Consider the following IDL:</li></ol> <pre>// IDL interface example {     typedef sequence&lt;long&gt; unboundedseq;     unboundedseq myop(); };</pre>
---------	--

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

**Example 35:** *The idlmembername Copybook (Sheet 1 of 2)*

```
*****  
* Operation:      myop  
* Mapped name:    myop  
* Arguments:      None
```

Example 35: The idlmembername Copybook (Sheet 2 of 2)

88	EXAMPLE-UNBOUNDEDSEQ	VALUE
	"IDL:example/unboundedseq:1.0".	
88	EXAMPLE	VALUE
	"IDL:example:1.0".	
01	EXAMPLE-TYPE-LENGTH	PICTURE S9(09)
		BINARY VALUE 28.

3. The following is an example of how to use SEQGET in a client or server program:

```
WORKING-STORAGE SECTION.
01 WS-MAX-ELEMENTS                PICTURE 9(09) BINARY
                                   VALUE 10.
01 WS-CURRENT-ELEMENT             PICTURE 9(09) BINARY
                                   VALUE 0.

CALL "ORBEXEC"    USING EXAMPLE-OBJ
                    EXAMPLE-OPERATION
                    EXAMPLE-MYOP-ARGS
                    EXAMPLE-USER-EXCEPTIONS.

SET WS-ORBEXEC TO TRUE.
PERFORM CHECK-STATUS.
* Get each of the 10 elements in the sequence.
PERFORM VARYING WS-CURRENT-ELEMENT
                FROM 1 BY 1 UNTIL
                WS-CURRENT-ELEMENT >
                SEQUENCE-LENGTH OF
                EXAMPLE-MYOP-ARGS
* Get the current element
CALL "SEQGET" USING RESULT-SEQUENCE OF
                    EXAMPLE-MYOP-ARGS
                    WS-CURRENT-ELEMENT
                    RESULT OF
                    RESULT-1 OF
                    EXAMPLE-MYOP-ARGS
SET WS-SEQGET TO TRUE
```

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.

---

# SEQSET

---

Synopsis	<pre>SEQSET(out <i>buffer</i> sequence-control-data,         in 9(09) BINARY element-number,         in <i>buffer</i> sequence-data) // Places the specified data into the specified element of an // unbounded sequence.</pre>
----------	---

---

Usage	Common to clients and servers.
-------	--------------------------------

---

Description	The <code>SEQSET</code> function copies the supplied data into the requested element of an unbounded sequence. You can set any element ranging between 1 and the maximum size of a sequence plus one. If the current maximum element plus one is set, the sequence is then reallocated, to hold the enlarged sequence.
-------------	--

**Note:** You can call `SEQSET` only on unbounded sequences.

---

Parameters	The parameters for <code>SEQSET</code> can be described as follows:
sequence-control-data	This is an <code>in</code> parameter that contains the unbounded sequence control data.
element-number	This is an <code>in</code> parameter that specifies the index of the element number that is to be set.
sequence-data	This is an <code>in</code> parameter that contains the address of the buffer containing the data that is to be placed in the sequence.

---

Example	<div><div>1. Consider the following IDL:</div><pre>// IDL interface example {     typedef sequence&lt;long&gt; unboundedseq;     unboundedseq myop(); };</pre></div>
---------	--



- Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

**Example 36:** *The idlmembername Copybook (Sheet 1 of 2)*

```
*****
* Operation:      myop
* Mapped name:    myop
* Arguments:      None
* Returns:        example/unboundedseq
* User Exceptions: none
*****
01 EXAMPLE-MYOP-ARGS.
  03 RESULT-1.
    05 RESULT                                PICTURE S9(10) BINARY.
  03 RESULT-SEQUENCE.
    05 SEQUENCE-MAXIMUM                      PICTURE 9(09) BINARY
                                             VALUE 0.
    05 SEQUENCE-LENGTH                      PICTURE 9(09) BINARY
                                             VALUE 0.
    05 SEQUENCE-BUFFER                      POINTER
                                             VALUE NULL.
    05 SEQUENCE-TYPE                        POINTER
                                             VALUE NULL.
*****
*
* Operation List section
* This lists the operations and attributes which an
* interface supports
*
*****
01 EXAMPLE-OPERATION                      PICTURE X(21).
   88 EXAMPLE-MYOP                        VALUE
      "myop:IDL:example:1.0".
01 EXAMPLE-OPERATION-LENGTH              PICTURE 9(09) BINARY
                                             VALUE 21.
*****
*
* Typecode section
* This contains CDR encodings of necessary typecodes.
*
*****
01 EXAMPLE-TYPE                          PICTURE X(28).
   COPY CORBATYP.
```

**Example 36:** *The idlmembername Copybook (Sheet 2 of 2)*

88	EXAMPLE-UNBOUNDEDSEQ	VALUE
	"IDL:example/unboundedseq:1.0".	
88	EXAMPLE	VALUE
	"IDL:example:1.0".	
01	EXAMPLE-TYPE-LENGTH	PICTURE S9(09)
		BINARY VALUE 28.

3. The following is an example of how to use SEQSET in a client or server program:

**Example 37:** *Using SEQSET in Client or Server (Sheet 1 of 2)*

```
WORKING-STORAGE SECTION.
01 WS-MAX-ELEMENTS                PICTURE 9(09) BINARY
                                   VALUE 10.
01 WS-CURRENT-ELEMENT             PICTURE 9(09) BINARY
                                   VALUE 0.

DO-EXAMPLE-MYOP.
  CALL "COAGET" USING EXAMPLE-MYOP-ARGS.
  SET WS-COAGET TO TRUE.
  PERFORM CHECK-STATUS.
* initialize the maximum and length fields.

*      MOVE WS-MAX-ELEMENTS TO SEQUENCE-MAXIMUM OF
      MOVE 0                TO SEQUENCE-MAXIMUM OF
                                   EXAMPLE-MYOP-ARGS.
      MOVE 0                TO SEQUENCE-LENGTH OF
                                   EXAMPLE-MYOP-ARGS.

* Initialize the sequence element data
      MOVE 0 TO RESULT OF
                                   RESULT-1 OF
                                   EXAMPLE-MYOP-ARGS.
* set the typecode of the sequence
      SET EXAMPLE-UNBOUNDEDSEQ TO TRUE.
* Allocate memory for the unbounded sequence.
* NOTE: SEQUENCE-MAXIMUM is set to WS-MAX-ELEMENTS
* after SEQALLOC call.
      CALL "SEQALLOC" USING WS-MAX-ELEMENTS
                                   EXAMPLE-TYPE
                                   EXAMPLE-TYPE-LENGTH
                                   RESULT-SEQUENCE OF
                                   EXAMPLE-MYOP-ARGS.
```

**Example 37: Using SEQSET in Client or Server (Sheet 2 of 2)**

```

      SET WS-SEQALLOC TO TRUE.
      PERFORM CHECK-STATUS.
* Set each of the 10 elements in the sequence.
      PERFORM VARYING WS-CURRENT-ELEMENT
                    FROM 1 BY 1 UNTIL
                    WS-CURRENT-ELEMENT >
                    SEQUENCE-MAXIMUM OF
                    EXAMPLE-MYOP-ARGS
* initialize the element data
      ADD 2 TO      RESULT OF
                    RESULT-1 OF
                    EXAMPLE-MYOP-ARGS
      DISPLAY "Element data value equals "
                    RESULT OF
                    RESULT-1 OF
                    EXAMPLE-MYOP-ARGS

* Set the current element to the element data buffer
* NOTE: SEQUENCE-LENGTH is incremented on each seqset
      CALL "SEQSET" USING RESULT-SEQUENCE OF
                    EXAMPLE-MYOP-ARGS
                    WS-CURRENT-ELEMENT
                    RESULT OF
                    RESULT-1 OF
                    EXAMPLE-MYOP-ARGS

      SET WS-SEQSET TO TRUE
      PERFORM CHECK-STATUS
      END-PERFORM.

      CALL "COAPUT" USING EXAMPLE-MYOP-ARGS.
      SET WS-COAPUT TO TRUE.
      PERFORM CHECK-STATUS.

*****
* Check Errors Copybook
*****
COPY CHKERRS.

```

**Note:** The COPY CHKERRS statement in the preceding example is used in batch programs. It is replaced with COPY CERRSMFA in IMS or CICS server programs.

## 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.

---

# STRFREE

---

## Synopsis

```
STRFREE(in POINTER string-pointer)
// Frees the memory allocated to a bounded 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. Refer to [“Memory Handling” on page 383](#) for more details.

---

## Parameters

The parameters 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 bounded string.

---

## Example

The example can be broken down as follows:

1. Consider the following IDL:

```
interface sample {
    typedef string astring;
    attribute astring mystring;
};
```

- Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

```
*****
* Attribute:   mystring
* Mapped name: mystring
* Type:       sample/astring (read/write)
*****

01 SAMPLE-MYSTRING-ARGS.
    03 RESULT                                POINTER
                                           VALUE NULL.
```

- The following is an example of how to use `STRFREE` in a client or server program:

```
PROCEDURE DIVISION.
...
* note the string pointer will have been set
* by a call to STRSET/STRSETP
    CALL "STRFREE" USING RESULT OF SAMPLE-MYSTRING-ARGS.

    DISPLAY "The memory is now released".
```

See also

["STRSET" on page 362.](#)

---

# STRGET

Synopsis	<pre>STRGET(in POINTER string-pointer,       in 9(09) BINARY string-length,       out X(nn) string) // Copies the contents of an unbounded string to a bounded string.</pre>						
Usage	Common to clients and servers.						
Description	<p>The <code>STRGET</code> function copies the characters in the unbounded string pointer, <code>string-pointer</code>, to the <code>string</code> item. If the <code>string-pointer</code> parameter does not contain enough characters to exactly fill the target string, the target string is terminated by a space. If there are too many characters in the <code>string-pointer</code>, the excess characters are not copied to the target string.</p> <div><b>Note:</b> Null characters are never copied from the <code>string-pointer</code> to the target string.</div> <p>The number of characters copied depends on the length parameter. This must be a valid positive integer (that is, greater than zero); otherwise, a runtime error occurs. If the <code>X(nn)</code> data item is shorter than the length field, the string is still copied, but obviously cannot contain the intended string.</p>						
Parameters	<p>The parameters for <code>STRGET</code> can be described as follows:</p> <table><tr><td><code>string-pointer</code></td><td>This is an <code>in</code> parameter that is the unbounded string pointer containing a copy of the unbounded string.</td></tr><tr><td><code>string-length</code></td><td>This is an <code>in</code> parameter that specifies the length of the unbounded string.</td></tr><tr><td><code>string</code></td><td>This is an <code>out</code> 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.</td></tr></table>	<code>string-pointer</code>	This is an <code>in</code> parameter that is the unbounded string pointer containing a copy of the unbounded string.	<code>string-length</code>	This is an <code>in</code> parameter that specifies the length of the unbounded string.	<code>string</code>	This is an <code>out</code> 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.
<code>string-pointer</code>	This is an <code>in</code> parameter that is the unbounded string pointer containing a copy of the unbounded string.						
<code>string-length</code>	This is an <code>in</code> parameter that specifies the length of the unbounded string.						
<code>string</code>	This is an <code>out</code> 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.						

# Example

The example can be broken down as follows:

1. Consider the following IDL:

```
// IDL
interface sample
{
    typedef string astring;
    attribute astring mystring;
};
```

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

```
*****
* Attribute:    mystring
* Mapped name:  mystring
* Type:        sample/astring (read/write)
*****

01 SAMPLE-MYSTRING-ARGS.
03 RESULT                                POINTER
                                         VALUE NULL.
```



3. The following is an example of how to use `STRGET` in a client or server program:

```

WORKING-STORAGE SECTION.

01 WS-BOUNDED-STRING      PICTURE X(20) VALUE SPACES.
01 WS-BOUNDED-STRING-LEN  PICTURE 9(09) BINARY VALUE 20.

PROCEDURE DIVISION.

* note the string pointer will have been set
* by a call to STRSET/STRSETP
...
    CALL "STRGET" USING RESULT OF MYSTRING-ARGS
                        WS-BOUNDED-STRING-LEN
                        WS-BOUNDED-STRING.

    SET WS-STRGET TO TRUE.
    PERFORM CHECK-STATUS.
    DISPLAY "Bounded string now retrieved and value equals "
           WS-BOUNDED-STRING.

```

---

# STRLEN

Synopsis	<pre>STRLEN(in POINTER string-pointer,       out 9(09) BINARY string-length) // Returns the actual length of an unbounded string.</pre>
Usage	Common to clients and servers.
Description	The <code>STRLEN</code> function returns the number of characters in an unbounded string.
Parameters	<p>The parameters for <code>STRLEN</code> can be described as follows:</p> <p><code>string-pointer</code> This is an <code>in</code> parameter that is the unbounded string pointer containing the unbounded string.</p> <p><code>string-length</code> This is an <code>out</code> parameter that is used to retrieve the actual length of the string that the <code>string-pointer</code> contains.</p>
Example	<p>The example can be broken down as follows:</p> <ol style="list-style-type: none"><li>1. Consider the following IDL:</li></ol> <pre>// IDL interface sample {     typedef string astring;     attribute astring mystring; };</pre>

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

```
*****
* Attribute:   mystring
* Mapped name: mystring
* Type:       sample/astring (read/write)
*****

01 SAMPLE-MYSTRING-ARGS.
03 RESULT                                POINTER
                                         VALUE NULL.
```

3. The following is an example of how to use `STRLEN` in a client or server program:

```
WORKING-STORAGE SECTION.

01 WS-BOUNDED-STRING-LEN    PICTURE 9(09) BINARY VALUE 0.

PROCEDURE DIVISION.
    ...
    * note the string pointer will have been set
    * by a call to STRSET/STRSETP
    CALL "STRLEN" USING RESULT OF MYSTRING-ARGS
                                WS-BOUNDED-STRING-LEN.

    DISPLAY "The String length equals  set".
    WS-BOUNDED-STRING-LEN
```

# STRSET

<p><b>Synopsis</b></p>	<pre>STRSET(out POINTER string-pointer,        in 9(09) BINARY string-length,        in X(nn) string) // Creates a dynamic string from a PIC X(n) data item</pre>
<p><b>Usage</b></p>	<p>Common to clients and servers</p>
<p><b>Description</b></p>	<p>The <code>STRSET</code> function creates an unbounded string to which it copies the number of characters specified in <code>length</code> from the bounded string specified in <code>string</code>. If the bounded string contains trailing spaces, these are not copied to the target unbounded string whose memory location is specified by <code>string-pointer</code>.</p> <p>The <code>STRSETP</code> version of this function is identical, except that it does copy trailing spaces. You can use the <code>STRFREE</code> to subsequently free this allocated memory.</p> <p>The number of characters copied depends on the <code>length</code> parameter. This must be a valid positive integer (that is, greater than zero); otherwise, a runtime error occurs. If the <code>X(nn)</code> data item is shorter than the <code>length</code> field, the string is still copied, but obviously cannot contain the intended string.</p> <div data-bbox="494 1012 1253 1142"> <p><b>Note:</b> <code>STRSET</code> allocates memory for the string from the program heap at runtime. Refer to <a href="#">“STRFREE” on page 355</a> and <a href="#">“Unbounded Strings and Memory Management” on page 389</a> for details of how this memory is subsequently released.</p> </div>
<p><b>Parameters</b></p>	<p>The parameters for <code>STRSET</code> can be described as follows:</p> <p><code>string-pointer</code> This is an <code>out</code> parameter to which the unbounded string is copied.</p> <p><code>string-length</code> This is an <code>in</code> parameter that specifies the number of characters to be copied from the bounded string specified in <code>string</code>.</p>

`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.

## Example

The example can be broken down as follows:

1. Consider the following IDL:

```
// IDL
interface sample
{
    typedef string astring;
    attribute astring mystring;
};
```

2. Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

```
*****
* Attribute:    mystring
* Mapped name: mystring
* Type:        sample/astring (read/write)
*****

01 SAMPLE-MYSTRING-ARGS.
03 RESULT                                POINTER
                                           VALUE NULL.
```

3. The following is an example of how to use `STRSET` in a client or server program:

```
WORKING-STORAGE SECTION.  
  
01 WS-BOUNDED-STRING          PICTURE X(20) VALUE SPACES.  
01 WS-BOUNDED-STRING-LEN      PICTURE 9(09) BINARY VALUE 20.  
  
PROCEDURE DIVISION.  
    ...  
    * Note trailing spaces are not copied.  
    MOVE "JOE BLOGGS" TO WS-BOUNDED-STRING.  
    CALL "STRSET" USING RESULT OF SAMPLE-MYSTRING-ARGS  
                        WS-BOUNDED-STRING-LEN  
                        WS-BOUNDED-STRING.  
  
    SET WS-STRSET TO TRUE.  
    PERFORM CHECK-STATUS.  
  
    DISPLAY "String pointer is now set".
```

---

**See also**

- [“STRFREE” on page 355.](#)
- [“Unbounded Strings and Memory Management” on page 389.](#)

---

# STRSETP

## Synopsis

```
STRSETP(out POINTER string-pointer,  
        in 9(09) BINARY string-length,  
        in X(nn) string)  
// Creates a dynamic string from a PIC X(n) data item.
```

## Usage

Common to clients and servers.

## Description

The `STRSETP` function is exactly the same as `STRSET`, except that `STRSETP` does copy trailing spaces to the unbounded string. Refer to [“STRSET” on page 362](#) for more details.

**Note:** `STRSETP` allocates memory for the string from the program heap at runtime. Refer to [“STRFREE” on page 355](#) and [“Unbounded Strings and Memory Management” on page 389](#) for details of how this memory is subsequently released.

## Example

The example can be broken down as follows

1. Consider the following IDL:

```
//IDL  
interface sample  
{  
    typedef string astring;  
    attribute astring mystring;  
};
```

- Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

```
*****
* Attribute:   mystring
* Mapped name: mystring
* Type:       sample/astring (read/write)
*****

01 SAMPLE-MYSTRING-ARGS.
03 RESULT                                POINTER
                                         VALUE NULL.
```

- The following is an example of how to use `STRSETP` in a client or server program:

```
WORKING-STORAGE SECTION.

01 WS-BOUNDED-STRING      PICTURE X(20) VALUE SPACES.
01 WS-BOUNDED-STRING-LEN  PICTURE 9(09) BINARY VALUE 20.

PROCEDURE DIVISION.
...
* Note trailing spaces are copied.
  MOVE "JOE BLOGGS" TO WS-BOUNDED-STRING.
  CALL "STRSETP" USING RESULT OF MYSTRING-ARGS
                      WS-BOUNDED-STRING-LEN
                      WS-BOUNDED-STRING.

  SET WS-STRSETP TO TRUE.
  PERFORM CHECK-STATUS.

  DISPLAY "String pointer is now set".
```

See also

- [“STRFREE” on page 355.](#)
- [“Unbounded Strings and Memory Management” on page 389.](#)



# STRTOOBJ

Synopsis	<pre> STRTOOBJ(in POINTER object-string,           out POINTER object-reference) // Creates an object reference from an interoperable object // reference (IOR). </pre>
Usage	<p>Common to clients and servers.</p>
Description	<p>The STRTOOBJ function creates an object reference from a stringfied IOR, by calling the ORB::string_to_object() C++ function.</p> <p>A stringfied interoperable object reference is of the form:</p> <pre> IOR:010000001c00000049444c3a53696d706c652f53696d706c654f626a6563 743a312e300001000000000000007e000000010102000a0000006a7578746 1706f736500e803330000003a3e023231096a75787461706f73651273696d 706c655f70657273697374656e7400106d795f73696d706c655f6f626a656 374000200000001000000180000000100000001000100000000000010100 01000000090101000600000006000000010000002100 </pre> <p>You can use the supplied <code>iordump</code> utility to parse the IOR. The <code>iordump</code> utility is available with Orbix E2A in UNIX System Services.</p>
Parameters	<p>The parameters for STRTOOBJ can be described as follows:</p> <div> <div>object-string</div> <div>This is an <code>in</code> parameter that contains a pointer to the address in memory where the interoperable object reference is held.</div> </div> <div> <div>object-reference</div> <div>This is an <code>out</code> parameter that contains a pointer to the address in memory where the returned object reference is held.</div> </div>

**Example**

The following is an example of how to use `STRTOOBJ` in a client or server program:

```
WORKING-STORAGE SECTION.
* Normally not stored in Working storage - this is just for
  demonstration.
01 WS-SIMPLE-IOR                PIC X(2048) VALUE
  "IOR:010000001c00000049444c3a53696d706c652f53696d706c654f626a
  6563743a312e3000010000000000000007e000000010102000a0000006a757
  87461706f736500e803330000003a3e023231096a75787461706f73651273
  696d706c655f70657273697374656e7400106d795f73696d706c655f6f626
  a65637400020000000100000018000000010000000100010000000000001
  0100010000000901010006000000060000000100000002100"
01 WS-SIMPLE-SIMPLEOBJECT      POINTER VALUE NULL.

* Set the COBOL pointer to point to the IOR string
* Normally read from a file
  CALL "STRSET"      USING IOR-REC-PTR
                        IOR-REC-LEN
                        WS-SIMPLE-IOR.

  SET WS-STRSET TO TRUE.
  PERFORM CHECK-STATUS.
* Obtain object reference from the IOR
  CALL "STRTOOBJ"    USING IOR-REC-PTR
                        WS-SIMPLE-SIMPLEOBJECT

  SET WS-STRTOOBJ TO TRUE.
  PERFORM CHECK-STATUS.
```

See also

[“OBJTOSTR” on page 312.](#)

---

# TYPEGET

## Synopsis

```
TYPEGET(inout POINTER any-pointer,
        in 9(09) BINARY typecode-key-length,
        out X(nn) typecode-key)
// 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:

<code>any-pointer</code>	This is an <code>inout</code> parameter that is a pointer to the address in memory where the <code>any</code> is stored.
<code>typecode-key-length</code>	This is an <code>in</code> parameter that specifies the length of the typecode key, as defined in the <i>idlmembername</i> copybook generated by the Orbix E2A IDL compiler.
<code>typecode-key</code>	This is an <code>out</code> parameter that contains a 01 level data item to which the typecode key is copied. This is defined in the <i>idlmembername</i> copybook generated by the Orbix E2A IDL compiler. This is a bounded string.

## Example

The example can be broken down as follows:

1. Consider the following IDL:

```
// IDL
interface sample
{
    attribute any myany;
};
```

- Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code code in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

```

01 SAMPLE-MYANY-ARGS.
    03 RESULT                                POINTER
                                           VALUE NULL.
...
01 EXAMPLE-TYPE                                PICTURE X(15).
    COPY CORBATYP.
    88 SAMPLE                                VALUE
        "IDL:sample:1.0".
01 EXAMPLE-TYPE-LENGTH                        PICTURE S9(09) BINARY
                                           VALUE 22.

```

- The following is an example of how to use TYPEGET in a client or server program:

```

WORKING-STORAGE SECTION.
    01 WS-DATA                                PIC S9(5) VALUE 0.

CALL "TYPEGET" USING RESULT OF SAMPLE-MYANY-ARGS
                                EXAMPLE-TYPE-LENGTH
                                EXAMPLE-TYPE.

SET WS-TYPEGET TO TRUE.
PERFORM CHECK-STATUS.
* validate typecode
EVALUATE TRUE
    WHEN CORBA-TYPE-SHORT
*retrieve the ANY CORBA::Short value
    CALL "ANYGET" USING RESULT OF SAMPLE-MYANY-ARGS
                                WS-DATA

    SET WS-ANYGET TO TRUE
    PERFORM CHECK-STATUS
    DISPLAY "ANY value equals " WS-DATA.
    WHEN OTHER
        DISPLAY "Wrong typecode received, expected a SHORT
                typecode "
END-EVALUATE.

```

## Exceptions

A `CORBA::BAD_INV_ORDER::TYPESET_NOT_CALLED` exception is raised if the typecode of the `any` has not been set via `TYPESET`.

---

# TYPESET

Synopsis	<pre>TYPESET(inout POINTER any-pointer,         in 9(09) BINARY typecode-key-length,         in X(nn) typecode-key) // Sets the type name of an any.</pre>						
Description	<p>The <code>TYPESET</code> function sets the type of the <code>any</code> to the supplied typecode. You must call <code>TYPESET</code> before you call <code>ANYSET</code>, because <code>ANYSET</code> uses the current typecode information to insert the data into the <code>any</code>.</p>						
Parameters	<p>The parameters for <code>TYPESET</code> can be described as follows:</p> <table><tr><td><code>any-type</code></td><td>This is an <code>inout</code> parameter that is a pointer to the address in memory where the <code>any</code> is stored.</td></tr><tr><td><code>typecode-key-length</code></td><td>This is an <code>in</code> parameter that specifies the length of the typecode string, as defined in the <i>idlmembername</i> copybook generated by the Orbix E2A IDL compiler.</td></tr><tr><td><code>typecode-key</code></td><td>This is an <code>in</code> parameter containing the typecode string representation, as defined in the <i>idlmembername</i> copybook generated by the Orbix E2A IDL compiler. The appropriate 88 level item is set for the typecode to be used.</td></tr></table>	<code>any-type</code>	This is an <code>inout</code> parameter that is a pointer to the address in memory where the <code>any</code> is stored.	<code>typecode-key-length</code>	This is an <code>in</code> parameter that specifies the length of the typecode string, as defined in the <i>idlmembername</i> copybook generated by the Orbix E2A IDL compiler.	<code>typecode-key</code>	This is an <code>in</code> parameter containing the typecode string representation, as defined in the <i>idlmembername</i> copybook generated by the Orbix E2A IDL compiler. The appropriate 88 level item is set for the typecode to be used.
<code>any-type</code>	This is an <code>inout</code> parameter that is a pointer to the address in memory where the <code>any</code> is stored.						
<code>typecode-key-length</code>	This is an <code>in</code> parameter that specifies the length of the typecode string, as defined in the <i>idlmembername</i> copybook generated by the Orbix E2A IDL compiler.						
<code>typecode-key</code>	This is an <code>in</code> parameter containing the typecode string representation, as defined in the <i>idlmembername</i> copybook generated by the Orbix E2A IDL compiler. The appropriate 88 level item is set for the typecode to be used.						
Example	<p>The example can be broken down as follows:</p> <ol style="list-style-type: none"><li>1. Consider the following IDL:</li></ol>						

- Based on the preceding IDL, the Orbix E2A IDL compiler generates the following code in the *idlmembername* copybook (where *idlmembername* represents the (possibly abbreviated) name of the IDL member that contains the IDL definitions):

```

01 SAMPLE-MYANY-ARGS.
    03 RESULT                                POINTER
                                           VALUE NULL.
*****
*
* Typecode section
* This contains CDR encodings of necessary typecodes.
*
*****

01 EXAMPLE-TYPE                                PICTURE X(15).
COPY CORBATYP.
    88 SAMPLE                                VALUE
        "IDL:sample:1.0".
01 EXAMPLE-TYPE-LENGTH                        PICTURE S9(09)
                                           BINARY VALUE 22.

```

- The following is an example of how to use TYPESET in a client or server program:

```

WORKING-STORAGE SECTION.
01 WS-DATA                                PIC S9(5) VALUE 0.

PROCEDURE DIVISION.
...
* Set the ANY typecode to be a CORBA::ShortLong
SET CORBA-TYPE-SHORT TO TRUE.
CALL "TYPESET"    USING RESULT OF
                    SAMPLE-MYANY-ARGS
                    EXAMPLE-TYPE-LENGTH
                    EXAMPLE-TYPE.

SET WS-TYPESET TO TRUE.
PERFORM CHECK-STATUS.

```

See also

- [“ANYFREE” on page 270.](#)
- [“The any Type and Memory Management” on page 397.](#)

---

# WSTRFREE

## Synopsis

---

```
WSTRFREE(in POINTER widestring-pointer)  
// Frees the memory allocated to a bounded wide string.
```

---

---

---

---

# WSTRGET

Synopsis	<pre>WSTRGET(in POINTER widestring-pointer,         in 9(09) BINARY widestring-length,         out G(nn) widestring) // Copies the contents of an unbounded wide string to a bounded // wide string.</pre>						
Usage	Common to clients and servers.						
Description	<p>The <code>WSTRGET</code> function copies the characters in the unbounded wide string pointer, <code>string_pointer</code>, to the COBOL <code>PIC X(n)</code> wide string item. If the <code>string_pointer</code> parameter does not contain enough characters to exactly fill the target wide string, the target wide string is terminated by a space. If there are too many characters in the <code>string_pointer</code>, the excess characters are not copied to the target wide string.</p> <div><b>Note:</b> Null characters are never copied from the <code>string_pointer</code> to the target wide string.</div>						
Parameters	<p>The parameters for <code>WSTRGET</code> can be described as follows:</p> <table><tr><td><code>widestring-pointer</code></td><td>This is an <code>in</code> parameter that is the unbounded wide string pointer containing a copy of the unbounded wide string.</td></tr><tr><td><code>widestring-length</code></td><td>This is an <code>in</code> parameter that specifies the length of the unbounded wide string.</td></tr><tr><td><code>widestring</code></td><td>This is an <code>out</code> parameter that is a bounded wide string to which the contents of the wide string pointer are copied. This wide string is terminated by a space if it is larger than the contents of the wide string pointer.</td></tr></table>	<code>widestring-pointer</code>	This is an <code>in</code> parameter that is the unbounded wide string pointer containing a copy of the unbounded wide string.	<code>widestring-length</code>	This is an <code>in</code> parameter that specifies the length of the unbounded wide string.	<code>widestring</code>	This is an <code>out</code> parameter that is a bounded wide string to which the contents of the wide string pointer are copied. This wide string is terminated by a space if it is larger than the contents of the wide string pointer.
<code>widestring-pointer</code>	This is an <code>in</code> parameter that is the unbounded wide string pointer containing a copy of the unbounded wide string.						
<code>widestring-length</code>	This is an <code>in</code> parameter that specifies the length of the unbounded wide string.						
<code>widestring</code>	This is an <code>out</code> parameter that is a bounded wide string to which the contents of the wide string pointer are copied. This wide string is terminated by a space if it is larger than the contents of the wide string pointer.						



---

# WSTRLEN

## Synopsis

---

```
WSTRLEN(in POINTER widestring-pointer,  
        out 9(09) BINARY widestring-length)  
// Returns the actual length of an unbounded wide string.
```

---

## 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:

<code>widestring-pointer</code>	This is an <code>in</code> parameter that is the unbounded wide string pointer containing the unbounded wide string.
<code>widestring-length</code>	This is an <code>out</code> parameter that is used to retrieve the actual length of the wide string that the <code>string-pointer</code> contains.

---

# WSTRSET

Synopsis	<pre>WSTRSET(out POINTER widestring-pointer,         in 9(09) BINARY widestring-length,         in G(nn) widestring) // Creates a dynamic wide string from a PIC G(n) data item</pre>						
Usage	Common to clients and servers						
Description	<p>The <code>WSTRSET</code> function creates an unbounded wide string to which it copies the number of characters specified in <code>length</code> from the bounded wide string specified in <code>string</code>. If the bounded wide string contains trailing spaces, these are not copied to the target unbounded wide string whose memory location is specified by <code>string-pointer</code>.</p> <p>The <code>WSTRSETP</code> version of this function is identical, except that it does copy trailing spaces. You can use the <code>WSTRFREE</code> to subsequently free this allocated memory.</p>						
Parameters	<p>The parameters for <code>WSTRSET</code> can be described as follows:</p> <table><tr><td><code>widestring-pointer</code></td><td>This is an <code>out</code> parameter to which the unbounded string is copied.</td></tr><tr><td><code>widestring-length</code></td><td>This is an <code>in</code> parameter that specifies the number of characters to be copied from the bounded string specified in <code>string</code>.</td></tr><tr><td><code>widestring</code></td><td>This is an <code>in</code> 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.</td></tr></table>	<code>widestring-pointer</code>	This is an <code>out</code> parameter to which the unbounded string is copied.	<code>widestring-length</code>	This is an <code>in</code> parameter that specifies the number of characters to be copied from the bounded string specified in <code>string</code> .	<code>widestring</code>	This is an <code>in</code> 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.
<code>widestring-pointer</code>	This is an <code>out</code> parameter to which the unbounded string is copied.						
<code>widestring-length</code>	This is an <code>in</code> parameter that specifies the number of characters to be copied from the bounded string specified in <code>string</code> .						
<code>widestring</code>	This is an <code>in</code> 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.						

---

# WSTRSETP

---

## Synopsis

```
WSTRSETP(out POINTER widestring-pointer,  
         in 9(09) BINARY widestring-length,  
         in G(nn) widestring)  
// Creates a dynamic wide string from a PIC G(n) data item.
```

---

## Usage

Common to clients and servers.

---

## Description

The `WSTRSETP` function is exactly the same as `WSTRSET`, except that `WSTRSETP` does copy trailing spaces to the unbounded wide string. Refer to [“WSTRSET” on page 376](#) for more details.

# CHECK-STATUS

Synopsis

CHECK-STATUS  
// Checks to see if a system exception has occurred on an API call.

Usage

Common to clients and servers.

Description

The CHECK-STATUS paragraph written in COBOL checks to see if a system exception has occurred on an API call. It is not an API in the COBOL runtime. It is contained in the *orbixhlq.INCLUDE.COPYLIB(CHKERRS)* member. To use CHECK-STATUS, you must use ORBSTAT to register the ORBIX-STATUS-INFORMATION block with the COBOL runtime. (Refer to [“ORBSTAT” on page 329.](#)) You should call CHECK-STATUS from the application on each subsequent API call, to determine if an exception has occurred on that API call.

The CHECK-STATUS paragraph checks the CORBA-EXCEPTION variable that is defined in the ORBIX-STATUS-INFORMATION block, and which is updated after every API call. If an exception has occurred, the following fields are set in the ORBIX-STATUS-INFORMATION block:

CORBA-EXCEPTION	This contains the appropriate value relating to the exception that has occurred. Values are in the range 1–36. A 0 value means no exception has occurred.
COMPLETION-STATUS-	This can be: COMPLETION-STATUS-YES—Value 0. COMPLETION-STATUS-NO—Value 1. COMPLETION-STATUS-MAYBE—Value 2.
EXCEPTION-TEXT	This is a COBOL pointer that contains a reference to the text of the CORBA system exception that has occurred.

**Note:** When an exception occurs, the JCL RETURN CODE is set to 12 and the application terminates.

**Parameters**

CHECK-STATUS takes no parameters.

**Definition**

The CHECK-STATUS function is defined as follows in the CHKERRS copybook:

```
CHECK-STATUS.
  IF NOT CORBA-NO-EXCEPTION THEN
    DISPLAY "Call Failed in api: " WS-API-CALLED
    SET CORBA-EXCEPTION-INDEX TO CORBA-EXCEPTION
    SET CORBA-EXCEPTION-INDEX UP BY 1
    DISPLAY "CORBA System Exception raised: "
      CORBA-EXCEPTION-NAME(CORBA-EXCEPTION-INDEX)
    SET CORBA-STATUS-INDEX TO COMPLETION-STATUS
    SET CORBA-STATUS-INDEX UP BY 1
    DISPLAY "Completion Status: "
      CORBA-STATUS-NAME(CORBA-STATUS-INDEX)

    CALL "STRGET" USING EXCEPTION-TEXT
                        ERROR-TEXT-LEN OF
                        ORBIX-EXCEPTION-TEXT
                        ERROR-TEXT OF
                        ORBIX-EXCEPTION-TEXT

    DISPLAY "CORBA Error Message: "
    DISPLAY ERROR-TEXT OF ORBIX-EXCEPTION-TEXT (1:64)
    DISPLAY ERROR-TEXT OF ORBIX-EXCEPTION-TEXT (64:64)
    DISPLAY ERROR-TEXT OF ORBIX-EXCEPTION-TEXT (128:64)
    MOVE 12 TO RETURN-CODE
    STOP RUN
  END-IF.
```

**Note:** The CHECK-STATUS paragraph in the CERRSMFA copybook is almost exactly the same, except it does not set the RETURN-CODE register, and it calls GOBACK instead of STOP RUN if a system exception occurs. This means that the native version of CHECK-STATUS is used to update the return code and exit the program.

**Example**

The following is an example of how to use `CHECK-STATUS` in the batch server implementation program:

```
DO-SIMPLE-SIMPLEOBJECT-CALL-ME.  
  CALL "COAGET" USING SIMPLE-SIMPLEOBJECT-70FE-ARGS.  
  SET WS-COAGET TO TRUE.  
  PERFORM CHECK-STATUS.  
  
  CALL "COAPUT" USING SIMPLE-SIMPLEOBJECT-70FE-ARGS.  
  SET WS-COAPUT TO TRUE.  
  PERFORM CHECK-STATUS.  
  
*****  
* Check Errors Copybook  
*****  
  
  COPY CHKERRS.
```

**Note:** The `COPY CHKERRS` statement in the preceding example is replaced with `COPY CERRSMFA` in the IMS or CICS server programs.

# Deprecated APIs

## Deprecated APIs

This section summarizes the APIs that were available with the Orbix 2.3 COBOL adapter, but which are now deprecated with the Orbix E2A COBOL runtime. It also outlines the APIs that are replacing these deprecated APIs.

```
OBJGET(IN object_ref, OUT dest_pointer, IN src_length)
// Orbix 2.3 : Returned a stringified Orbix object reference.
// Orbix E2A: No replacement. Supported on the server side for
//             migration purposes.
```

```
OBJGETI(IN object_ref, OUT dest_pointer, IN dest_length)
// Orbix 2.3 : Returned a stringified interoperable object
//             reference (IOR) from a valid object reference.
// Orbix E2A: Replaced by OBJTOSTR.
```

```
OBJSET(IN object_name, OUT object_ref)
// Orbix 2.3 : Created an object reference from a stringified
//             object reference.
// Orbix E2A: Replaced by STRTOOBJ.
```

```
OBJSETM(IN object_name, IN marker, OUT object_ref)
// Orbix 2.3 : Created an object reference from a stringified
//             object reference and set its marker.
// Orbix E2A: Replaced by OBJNEW.
```

```
ORBALLOC(IN length, OUT pointer)
// Orbix 2.3 : Allocated memory at runtime.
// Orbix E2A: Replaced by MEMALLOC.
```

```
ORBFREE(IN pointer)
// Orbix 2.3 : Freed memory.
// Orbix E2A: Replaced by MEMFREE and STRFREE.
```

```
ORBGET(INOUT complete_cobol_operation_parameter_buffer)
// Orbix 2.3 : Got IN and INOUT values.
// Orbix E2A: Replaced by COAGET.
```

```
ORBINIT(IN server_name, IN server_name_len)
// Orbix 2.3 : Equivalent to impl_is_ready in C++.
// Orbix E2A: Replaced by COARUN.
```

```

ORBPUT(INOUT complete_cobol_operation_parameter_buffer)
// Orbix 2.3 : Returned INOUT, OUT & result values.
// Orbix E2A: Replaced by COAPUT.

ORBREGO(IN cobol_interface_description, OUT object_ref)
// Orbix 2.3 : Describes an interface to the COBOL adapter and
//              creates an object reference using the interface
//              description.
// Orbix E2A: Replaced by OBJNEW and ORBREG.

ORBREQ(IN request_info_buffer)
// Orbix 2.3 : Provided current request information.
// Orbix E2A: Replaced by COAREQ.

STRSETSP(OUT dest_pointer, IN src_length, IN src)
// Orbix 2.3 : Created a dynamic string from a PIC X(n) data item.
// Orbix E2A: Replaced by STRSETP.

```



# 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 E2A COBOL application. It first describes in detail the memory handling rules adopted by the COBOL 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:

<a href="#">Operation Parameters</a>	<a href="#">page 384</a>
<a href="#">Memory Management Routines</a>	<a href="#">page 404</a>

---

# Operation Parameters

---

**Overview**

This section describes in detail the memory handling rules adopted by the COBOL 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:

<a href="#">Unbounded Sequences and Memory Management</a>	<a href="#">page 385</a>
<a href="#">Unbounded Strings and Memory Management</a>	<a href="#">page 389</a>
<a href="#">The any Type and Memory Management</a>	<a href="#">page 397</a>
<a href="#">Memory Management Routines</a>	<a href="#">page 404</a>

# Unbounded Sequences and Memory Management

Overview for IN parameters

Table 27 provides a detailed outline of how memory is handled for unbounded sequences that are used as `in` parameters.

Table 27: *Memory Handling for IN Unbounded Sequences*

Client Application	Server Application
1. SEQALLOC 2. SEQSET 3. OREXEC—(send)  7. SEQFREE	4. COAGET—(receive, allocate) 5. SEQGET 6. COAPUT—(free)

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 27:

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 `OREXEC`, which causes the client-side COBOL runtime to marshal the values across the network.
4. The server calls `COAGET`, which causes the server-side COBOL 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 `COAPUT`, which causes the server-side COBOL runtime to implicitly free the memory allocated by the call to `COAGET`.
7. The client calls `SEQFREE` to free the memory allocated by the call to `SEQALLOC`.

Overview for INOUT parameters

Table 28 provides a detailed outline of how memory is handled for unbounded sequences that are used as `inout` parameters.

**Table 28:** *Memory Handling for INOUT Unbounded Sequences*

Client Application	Server Application
1. SEQALLOC 2. SEQSET 3. OREXEC—(send)       10. (free, receive, allocate) 11. SEQGET 12. SEQFREE	4. COAGET—(receive, allocate) 5. SEQGET 6. SEQFREE 7. SEQALLOC 8. SEQSET 9. COAPUT—(send, free)

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 28:

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 `ORBEXEC`, which causes the client-side COBOL runtime to marshal the values across the network.
4. The server calls `COAGET`, which causes the server-side COBOL 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 `COAGET` 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 `SEQSET` to initialize the sequence elements for the replacement `out` sequence.
- 9. The server calls `COAPUT`, which causes the server-side COBOL 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 `ORBEXEC` in point 3 now causes the client-side COBOL runtime to:
  - vi. Free the memory allocated for the original `in` sequence via the call to `SEQALLOC` in point 1.
  - vii. Receive the replacement `out` sequence.
  - viii. Allocate memory for the replacement `out` sequence.

**Note:** By having `ORBEXEC` 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 `ORBEXEC` in point 3.

Overview for OUT and return parameters

Table 29 provides a detailed outline of how memory is handled for unbounded sequences that are used as `out` or `return` parameters.

Table 29: *Memory Handling for OUT and Return Unbounded Sequences*

Client Application	Server Application
1. ORBEXEC—(send)  6. (receive, allocate) 7. SEQGET 8. SEQFREE	2. COAGET—(receive) 3. SEQALLOC 4. SEQSET 5. COAPUT—(send, free)

**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 29](#):

1. The client calls `ORBEXEC`, which causes the client-side COBOL runtime to marshal the request across the network.
2. The server calls `COAGET`, which causes the server-side COBOL runtime to receive the client request.
3. The server calls `SEQALLOC` 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 `COAPUT`, which causes the server-side COBOL runtime to marshal the values across the network and implicitly free the memory allocated to the sequence via the call to `SEQALLOC`.
6. Control returns to the client, and the call to `ORBEXEC` in point 1 now causes the client-side COBOL 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 COBOL runtime to free the memory allocated for the sequence via the call to `ORBEXEC`.

# Unbounded Strings and Memory Management

Overview for IN parameters

Table 30 provides a detailed outline of how memory is handled for unbounded strings that are used as `in` parameters.

Table 30: *Memory Handling for IN Unbounded Strings*

Client Application	Server Application
1. STRSET 2. ORBEXEC—(send)  6. STRFREE	3. COAGET—(receive, allocate) 4. STRGET 5. COAPUT—(free)

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 30:

1. The client calls `STRSET` to initialize the unbounded string and allocate memory for it.
2. The client calls `ORBEXEC`, which causes the client-side COBOL runtime to marshal the values across the network.
3. The server calls `COAGET`, which causes the server-side COBOL 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 `COAPUT`, which causes the server-side COBOL runtime to implicitly free the memory allocated by the call to `COAGET`.
6. The client calls `STRFREE` to free the memory allocated by the call to `STRSET`.

Overview for INOUT parameters

[Table 31](#) provides a detailed outline of how memory is handled for unbounded strings that are used as `inout` parameters.

**Table 31:** *Memory Handling for INOUT Unbounded Strings*

Client Application	Server Application
1. STRSET 2. ORBEXEC—(send)  8. (free, receive, allocate) 9. STRGET 10. STRFREE	3. COAGET—(receive, allocate) 4. STRGET 5. STRFREE 6. STRSET 7. COAPUT—(send, free)

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 31](#):

1. The client calls `STRSET` to initialize the unbounded string and allocate memory for it.
2. The client calls `ORBEXEC`, which causes the client-side COBOL runtime to marshal the values across the network.
3. The server calls `COAGET`, which causes the server-side COBOL 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 `COAGET` in point 3.
6. The server calls `STRSET` to initialize the replacement `out` string and allocate memory for it.
7. The server calls `COAPUT`, which causes the server-side COBOL 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 `ORBEXEC` in point 2 now causes the client-side COBOL 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 `ORBEXEC` 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 `ORBEXEC` in point 2.

Overview for OUT and return parameters

Table 32 provides a detailed outline of how memory is handled for unbounded strings that are used as `out` or `return` parameters.

Table 32: Memory Handling for OUT and Return Unbounded Strings

Client Application	Server Application
1. ORBEXEC—(send)  5. (receive, allocate) 6. STRGET 7. STRFREE	2. COAGET—(receive) 3. STRSET 4. COAPUT—(send, free)

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 32:
- 1. The client calls `ORBEXEC`, which causes the client-side COBOL runtime to marshal the request across the network.
  - 2. The server calls `COAGET`, which causes the server-side COBOL runtime to receive the client request.

3. The server calls `STRSET` to initialize the string and allocate memory for it.
4. The server calls `COAPUT`, which causes the server-side COBOL 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 `ORBEXEC` in point 1 now causes the client-side COBOL 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 COBOL runtime to free the memory allocated for the string in point 5 via the call to `ORBEXEC` in point 1.

# Object References and Memory Management

Overview for IN parameters

Table 33 provides a detailed outline of how memory is handled for object references that are used as `in` parameters.

Table 33: *Memory Handling for IN Object References*

Client Application	Server Application
1. Attain object reference 2. ORBEXEC—(send)  6. OBJREL	3. COAGET—(receive) 4. read 5. COAPUT

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 33:

1. The client attains an object reference through some retrieval mechanism (for example, by calling `STRTOOBJ` or `OBJRIR`).
2. The client calls `ORBEXEC`, which causes the client-side COBOL runtime to marshal the object reference across the network.
3. The server calls `COAGET`, which causes the server-side COBOL runtime to receive the object reference.
4. The server can now invoke on the object reference.
5. The server calls `COAPUT`, which causes the server-side COBOL runtime to implicitly free any memory allocated by the call to `COAGET`.
6. The client calls `OBJREL` to release the object.

Overview for INOUT parameters

Table 34 provides a detailed outline of how memory is handled for object references that are used as `inout` parameters.

**Table 34:** *Memory Handling for INOUT Object References*

Client Application	Server Application
1. Attain object reference 2. ORBEXEC—(send)       9. (receive) 10. read 11. OBJREL	3. COAGET—(receive) 4. read 5. OBJREL 6. Attain object reference 7. OBJDUP 8. COAPUT—(send)

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 34:

1. The client attains an object reference through some retrieval mechanism (for example, by calling `STRTOOBJ` or `OBJRIR`).
2. The client calls `ORBEXEC`, which causes the client-side COBOL runtime to marshal the object reference across the network.
3. The server calls `COAGET`, which causes the server-side COBOL 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 `STRTOOBJ` or `OBJRIR`).
7. The server calls `OBJDUP` to increment the object reference count and to prevent the call to `COAPUT` in point 8 from causing the replacement `out` object reference to be released.
8. The server calls `COAPUT`, which causes the server-side COBOL runtime to marshal the replacement `out` object reference across the network.

- 9. Control returns to the client, and the call to `ORBEXEC` in point 2 now causes the client-side COBOL runtime to receive the replacement `out` object reference.
- 10. The client can now invoke on the replacement object reference.
- 11. The client calls `OBJREL` to release the object.

Overview for OUT and return parameters

[Table 35](#) provides a detailed outline of how memory is handled for object references that are used as `out` or `return` parameters.

**Table 35:** *Memory Handling for OUT and Return Object References*

Client Application	Server Application
1. ORBEXEC—(send)     6. (receive) 7. read 8. OBJREL	2. COAGET—(receive) 3. Attain object reference 4. OBJDUP 5. COAPUT—(send)

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 35](#):

- 1. The client calls `ORBEXEC`, which causes the client-side COBOL runtime to marshal the request across the network.
- 2. The server calls `COAGET`, which causes the server-side COBOL runtime to receive the client request.
- 3. The server attains an object reference through some retrieval mechanism (for example, by calling `STRTOOBJ` or `OBJRIR`).
- 4. The server calls `OBJDUP` to increment the object reference count and to prevent the call to `COAPUT` in point 5 from causing the object reference to be released.
- 5. The server calls `COAPUT`, which causes the server-side COBOL runtime to marshal the object reference across the network.
- 6. Control returns to the client, and the call to `ORBEXEC` in point 1 now causes the client-side COBOL 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 36 provides a detailed outline of how memory is handled for an `any` type that is used as an `in` parameter.

Table 36: *Memory Handling for IN Any Types*

Client Application	Server Application
1. TYPESET 2. ANYSET 3. ORBEXEC—(send)  8. ANYFREE	4. COAGET—(receive, allocate) 5. TYPEGET 6. ANYGET 7. COAPUT—(free)

Summary of rules for IN parameters

The memory handling rules for an `any` type used as an `in` parameter can be summarized as follows, based on Table 36:

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 `ORBEXEC`, which causes the client-side COBOL runtime to marshal the values across the network.
4. The server calls `COAGET`, which causes the server-side COBOL 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 `COAPUT`, which causes the server-side COBOL runtime to implicitly free the memory allocated by the call to `COAGET`.
8. The client calls `ANYFREE` to free the memory allocated by the call to `ANYSET`.

Overview for INOUT parameters

Table 37 provides a detailed outline of how memory is handled for an `any` type that is used as an `inout` parameter.

Table 37: *Memory Handling for INOUT Any Types*

Client Application	Server Application
1. TYPESET 2. ANYSET 3. ORBEXEC—(send)       11. (free, receive, allocate) 12. TYPEGET 13. ANYGET 14. ANYFREE	4. COAGET—(receive, allocate) 5. TYPEGET 6. ANYGET 7. ANYFREE 8. TYPSET 9. ANYSET 10. COAPUT—(send, free)

Summary of rules for INOUT parameters

The memory handling rules for an `any` type used as an `inout` parameter can be summarized as follows, based on Table 37:

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 `ORBEXEC`, which causes the client-side COBOL runtime to marshal the values across the network.
4. The server calls `COAGET`, which causes the server-side COBOL 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 `COAGET` 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 `COAPUT`, which causes the server-side COBOL 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 `ORBEEXEC` in point 3 now causes the client-side COBOL 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 `ORBEEXEC` 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 `ORBEEXEC` in point 3.

Overview for OUT and return parameters

Table 38 provides a detailed outline of how memory is handled for an `any` type that is used as an `out` or `return` parameter.

Table 38: *Memory Handling for OUT and Return Any Types*

Client Application	Server Application
1. ORBEXEC—(send)  6. (receive, allocate) 7. TYPEGET 8. ANYGET 9. ANYFREE	2. COAGET—(receive) 3. TYPESET 4. ANYSET 5. COAPUT—(send, free)

Summary of rules for OUT and return parameters

- The memory handling rules for an `any` type used as an `out` or `return` parameter can be summarized as follows, based on Table 38:
1. The client calls `ORBEXEC`, which causes the client-side COBOL runtime to marshal the request across the network.
  2. The server calls `COAGET`, which causes the server-side COBOL 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 `COAPUT`, which causes the server-side COBOL 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 `ORBEXEC` in point 1 now causes the client-side COBOL 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 `any` from the operation parameter buffer.

9. The client calls `ANYFREE`, which causes the client-side COBOL runtime to free the memory allocated for the `any` in point 6 via the call to `ORBEXEC` in point 1.

# User Exceptions and Memory Management

Overview

Table 39 provides a detailed outline of how memory is handled for user exceptions.

Table 39: Memory Handling for User Exceptions

Client Application	Server Application
1. ORBEXEC—(send)     6. Free	2. COAGET—(receive, allocate) 3. write 4. COAERR 5. (free)

Summary of rules

The memory handling rules for raised user exceptions can be summarized as follows, based on Table 39:

1. The client calls ORBEXEC, which causes the COBOL runtime to marshal the client request across the network.
2. The server calls COAGET, which causes the server-side COBOL 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 COAERR, to raise the user exception.
5. The server-side COBOL runtime automatically frees the memory allocated for the user exception in point 3.

**Note:** The COBOL 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 COBOL runtime or allocated manually. This should be done before the server calls COAERR.

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 COBOL 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 COBOL.

**Unbounded strings** Use `STRSET` to allocate memory for unbounded strings, and `STRFREE` to subsequently free this memory. For example:

```
01 MY-COBOL-STRING      PICTURE X(11) VALUE "Testing 123".
01 MY-COBOL-STRING-LEN  PIC 9(09) BINARY VALUE 11.
01 MY-CORBA-STRING      POINTER VALUE NULL.

* Allocation
CALL "STRSET"           USING MY-CORBA-STRING
                           MY-COBOL-STRING-LEN
                           MY-CORBA-STRING.

* Deletion
CALL "STRFREE"          USING MY-CORBA-STRING.
```

**Note:** Unbounded strings are stored internally as normal C or C++ strings that are terminated by a null character. The `STRx` routines provide facilities for copying these strings without the null character. The `STRx` routines also provide facilities for correctly truncating and padding the strings to and from their COBOL equivalents. It can be useful to know exactly how big the string actually is before copying it. You can use the `STRLEN` function to obtain this information.

**Unbounded wide strings**

Use `WSTRSET` to allocate memory for unbounded wide strings, and `WSTRFREE` to subsequently free this memory. For example:

```
01 MY-CORBA-WSTRING          POINTER VALUE NULL.

* Allocation
CALL "WSTRSET"      USING MY-CORBA-WSTRING
                      MY-COBOL-WSTRING-LEN
                      MY-CORBA-WSTRING.

* Deletion
CALL "WSTRFREE"     USING MY-CORBA-WSTRING.
```

**Typecodes**

As described in the Mapping chapter, typecodes are mapped to a pointer. They are handled in COBOL as unbounded strings and should contain a value corresponding to one of the typecode keys generated by the Orbix E2A IDL compiler. For example:

```
01 MY-TYPECODE              POINTER VALUE NULL.

* Allocation
CALL "STRSET"      USING MY-TYPECODE
                      MY-COMPLEX-TYPE
                      MY-COMPLEX-TYPE-LENGTH.

* Deletion
CALL "STRFREE"     USING MY-TYPECODE.
```

**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. You can also use `SEQSET` to resize the sequence if the maximum size of the sequence is not large enough to contain another sequence element. Use `SEQFREE` to free memory allocated via `SEQALLOC`. For example:

```
* Allocation
CALL "SEQALLOC"      USING MY-SEQUENCE-MAXIMUM
                      MY-USEQ-TYPE
                      MY-USEQ-TYPE-LENGTH
                      N-SEQUENCE OF MY-USEQ-ARGS.

* Deletion
CALL "SEQFREE"       USING N-SEQUENCE OF MY-USEQ-ARGS.
```

**Note:** You only need to call `SEQFREE` on the outermost sequence, because it automatically deletes both the sequence information block and any associated inner dynamic structures.

## 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:

```
01 MY-CORBA-ANY          POINTER VALUE NULL.
01 MY-LONG               PIC 9(10) BINARY VALUE 123.
* Allocation
SET CORBA-TYPE-LONG TO TRUE.
CALL "TYPESET"          USING MY-CORBA-ANY
                           MY-COMPLEX-TYPE-LENGTH
                           MY-COMPLEX-TYPE.

CALL "ANYSET"           USING MY-CORBA-ANY
                           MY-LONG.

* Deletion
CALL "ANYFREE"          USING MY-CORBA-ANY.
```



# POA Policies

*This chapter summarizes the POA policies that are supported by the Orbix E2A COBOL runtime, and the argument used with each policy.*

**In this chapter**

This chapter discusses the following topics:

<a href="#">Overview</a>	<a href="#">page 408</a>
<a href="#">Summary</a>	<a href="#">page 408</a>

## 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 E2A COBOL 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 in general. 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 E2A COBOL runtime supports. Orbix E2A COBOL programmers have no control over these POA policies. They are outlined here simply for the purposes of illustration and the sake of completeness.

## Summary

[Table 40](#) describes the policies that are supported by the Orbix E2A COBOL runtime, and the argument used with each policy.

**Table 40:** *POA Policies Supported by the Orbix E2A COBOL Runtime*

Policy	Argument Used	Description
Id Assignment	<code>USER_ID</code>	This policy determines whether object IDs are generated by the POA or the application. The <code>USER_ID</code> 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.  <code>USER_ID</code> is usually assigned to a POA that has an object lifespan policy of <code>PERSISTENT</code> (that is, it generates object references whose validity can span multiple instances of a POA or server process, so the application requires explicit control over object IDs).
Id Uniqueness	<code>MULTIPLE_ID</code>	This policy determines whether a servant can be associated with multiple objects in this POA. The <code>MULTIPLE_ID</code> specifies that any servant in the POA can be associated with multiple object IDs.

**Table 40:** *POA Policies Supported by the Orbix E2A COBOL Runtime*

Policy	Argument Used	Description
Implicit Activation	<code>NO_IMPLICIT_ACTIVATION</code>	This policy determines the POA's activation policy. The <code>NO_IMPLICIT_ACTIVATION</code> argument specifies that the POA only supports explicit activation of servants.
Lifespan	<code>PERSISTENT</code>	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.
Request Processing	<code>USE_ACTIVE_OBJECT_MAP_ONLY</code>	<p>This policy determines how the POA finds servants to implement requests. The <code>USE_ACTIVE_OBJECT_MAP_ONLY</code> 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 <code>deactivate_object()</code>.</p> <p>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 <code>OBJECT_NOT_EXIST</code> exception is raised to the client. This policy requires that the POA has a servant retention policy of <code>RETAIN</code>.</p>
Servant Retention	<code>RETAIN</code>	The <code>RETAIN</code> argument with this policy specifies that the POA retains active servants in its active object map.

**Table 40:** *POA Policies Supported by the Orbix E2A COBOL Runtime*

Policy	Argument Used	Description
Thread	<code>SINGLE_THREAD_MODEL</code>	The <code>SINGLE_THREAD_MODEL</code> argument with this policy specifies that requests for a single-threaded POA are processed sequentially. In a multi-threaded environment, all calls by a single-threaded POA to implementation code (that is, servants and servant managers) are made in a manner that is safe for code that does not account for multi-threading.

# System Exceptions

*This chapter summarizes the Orbix E2A system exceptions that are specific to the Orbix E2A COBOL runtime.*

**Note:** This chapter does not describe other Orbix E2A system exceptions that are not specific to the COBOL runtime. See the *CORBA Programmer's Guide, C++* for details of these other system exceptions.

**In this chapter**

This chapter discusses the following topics:

<a href="#">CORBA::INITIALIZE:: exceptions</a>	<a href="#">page 412</a>
<a href="#">CORBA::BAD_PARAM:: exceptions</a>	<a href="#">page 412</a>
<a href="#">CORBA::INTERNAL:: exceptions</a>	<a href="#">page 412</a>
<a href="#">CORBA::BAD_INV_ORDER:: exceptions</a>	<a href="#">page 413</a>
<a href="#">CORBA::DATA_CONVERSION:: exceptions</a>	<a href="#">page 413</a>

**CORBA::INITIALIZE::  
exceptions**

---

The following exception is defined within the `CORBA::INITIALIZE::` scope:

<code>UNKNOWN</code>	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:

<code>UNKNOWN_OPERATION</code>	This exception is raised by <code>ORBEXEC</code> , if the operation is not valid for the interface.
<code>NO_OBJECT_IDENTIFIER</code>	This exception is raised by <code>OBJNEW</code> , if the parameter for the object name is an invalid string.
<code>INVALID_SERVER_NAME</code>	This exception is raised if the server name that is passed does not match the server name passed to <code>ORBSRV</code> .

---

**CORBA::INTERNAL::  
exceptions**

The following exceptions are defined within the `CORBA::INTERNAL::` scope:

<code>UNEXPECTED_INVOCATION</code>	This exception is raised on the server side when a request is being processed, if a previous request has not completed successfully.
<code>UNKNOWN_TYPECODE</code>	This exception is raised internally by the COBOL 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 <i>idlmembernameX</i> copybook or the application itself.
<code>INVALID_STREAMABLE</code>	This exception is raised internally by the COBOL 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 <i>idlmembernameX</i> copybook or the application itself.

**CORBA::BAD\_INV\_ORDER::  
exceptions**

---

The following exceptions are defined within the CORBA::BAD\_INV\_ORDER:: scope:

INTERFACE_NOT_REGISTERED	This exception is raised if the specified interface has not been registered via ORBREG.
INTERFACE_ALREADY_REGISTERED	This exception is raised by ORBREG, if the client or server attempts to register the same interface more than once.
ADAPTER_ALREADY_INITIALIZED	This exception is raised by ORBARGS, if it is called more than once in a client or server.
STAT_ALREADY_CALLED	This exception is raised by ORBSTAT if it is called more than once.
SERVER_NAME_ALREADY_SET	This exception is raised by ORBSRV, if the API is called more than once.
SERVER_NAME_NOT_SET	This exception is raised by OBJNEW, COAREQ, OBJGETID, or COARUN, if ORBSRV is called.
NO_CURRENT_REQUEST	This exception is raised by COAREQ, if no request is currently in progress.
ARGS_NOT_READ	This exception is raised by COAPUT, if the in or inout parameters for the request have not been processed.
ARGS_ALREADY_READ	This exception is raised by COAGET, if the in or inout parameters for the request have already been processed.
TYPESET_NOT_CALLED	This exception is raised by ANYSET or TYPEGET, if the typecode for the any type has not been set via a call to TYPESET.

**CORBA::DATA\_CONVERSION::  
exceptions**

---

The following exception is defined within the CORBA::DATA\_CONVERSION:: scope:

VALUE_OUT_OF_RANGE	This exception is raised by ORBEXEC, COAGET, or COAPUT, if the value is determined to be out of range when marshalling a long, short, unsigned short, unsigned long long long, or unsigned long long type.
--------------------	--





# Index

## A

- abstract interfaces in IDL 139
- ADAPTER\_ALREADY\_INITIALIZED exception 413
- address space layout for COBOL application 47
- ANYFREE function 270
- ANYGET function 272
- ANYSET function 274
- any type
  - in IDL 142
  - mapping to COBOL 205
  - memory handling for 397
- APIs 263
- application interfaces, developing 19, 56, 89
- ARGS\_ALREADY\_READ exception 413
- ARGS\_NOT\_READ exception 413
- array type
  - in IDL 151
  - mapping to COBOL 203
- attributes
  - in IDL 126
  - mapping to COBOL 223

## B

- basic types
  - in IDL 141
  - mapping to COBOL 166
- bitwise operators 158
- boolean type, mapping to COBOL 171
- built-in types in IDL 141

## C

- char type
  - in IDL 142
  - mapping to COBOL 177
- CHECK-STATUS function 378
- client output 46
- clients
  - building 41, 82, 115
  - introduction to 5
  - running 45
  - writing 36, 77, 110
- COAERR function 277

- COAGET function 27, 65, 98, 281
- COAPUT function 27, 65, 98, 286
- COAREQ function 292
- COARUN function 297
- COBOL group data definitions 23, 60, 93
- COBOL runtime 6, 48, 263
- COBOL source, generating from IDL 21, 58, 91
- configuration domains 9
- constant definitions in IDL 155
- constant expressions in IDL 158
- constant fixed types in IDL 145
- copybooks, generating 21, 58, 91
- CORBA
  - introduction to 2
  - objects 3

## D

- data types, defining in IDL 154
- decimal fractions 145

## E

- empty interfaces in IDL 128
- enum type
  - in IDL 147
  - mapping to COBOL 174
  - ordinal values of 147
- exceptions, in IDL 127
  - See also* system exceptions, user exceptions
- extended built-in types in IDL 143

## F

- fixed type
  - in IDL 144
  - mapping to COBOL 187
- floating point type in IDL 141
- forward declaration of interfaces in IDL 134

## I

- Id Assignment policy 408
- identifier names, mapping to COBOL 163

## IDL

- abstract interfaces 139
- arrays 151
- attributes 126
- built-in types 141
- constant definitions 155
- constant expressions 158
- defining 20, 57, 90
- empty interfaces 128
- enum type 147
- exceptions 127
- extended built-in types 143
- forward declaration of interfaces 134
- inheritance redefinition 133
- interface inheritance 129
- introduction to interfaces 3
- local interfaces 135
- modules and name scoping 119
- multiple inheritance 130
- object interface inheritance 132
- operations 123
- pseudo object types 153
- sequence type 152
- struct type 148
- structure 118
- union type 149
- valuetypes 137

IDL-to-COBOL mapping

- any type 205
- array type 203
- attributes 223
- basic types 166
- boolean type 171
- char type 177
- enum type 174
- exception type 207
- fixed type 187
- identifier names 163
- object type 213
- octet type 179
- operations 218, 224
- sequence type 198
- string type 181
- struct type 191
- typedefs 210
- type names 165
- union type 193
- user exception type 207
- wide string type 186

- Id Uniqueness policy 408
- IIOF protocol 2
- Implicit Activation policy 409
- inheritance redefinition in IDL 133
- INTERFACE\_ALREADY\_REGISTERED exception 413
- interface inheritance in IDL 129
- INTERFACE\_NOT\_REGISTERED exception 413
- interfaces, developing for your application 19, 56, 89
- INVALID\_SERVER\_NAME exception 412
- INVALID\_STREAMABLE exception 412

## J

- JCL components, checking 18, 55, 88

## L

- Lifespan policy 409
- local interfaces in IDL 135
- local object pseudo-operations 136
- location domains 9
- locator daemon
  - introduction to 10
  - starting 43
- long double type in IDL 144
- long long type in IDL 143

## M

- MEMALLOC function 298
- MEMFREE function 300
- memory handling
  - any type 397
  - object references 393
  - routines for 404
  - unbounded sequences 385
  - unbounded strings 389
  - user exceptions 402
- modules and name scoping in IDL 119
- MULTIPLE\_ID argument 408
- multiple inheritance in IDL 130

## N

- NO\_CURRENT\_REQUEST exception 413
- node daemon
  - introduction to 10
  - starting 44
- NO\_IMPLICIT\_ACTIVATION argument 409

NO\_OBJECT\_IDENTIFIER exception 412

## O

OBJDUP function 301  
 object interface inheritance in IDL 132  
 object references  
   introduction to 3  
   memory handling for 393  
 object request broker. *See* ORB  
 objects, defined in CORBA 3  
 object type, mapping to COBOL 213  
 OBJGETI deprecated function 381  
 OBJGETID function 303  
 OBJNEW function 305  
 OBJREL function 308  
 OBJRIR function 310  
 OBJSET deprecated function 381  
 OBJTOSTR function 32, 312  
 octet type  
   in IDL 142  
   mapping to COBOL 179  
 operations  
   in IDL 123  
   mapping to COBOL 218  
 ORB, role of 5  
 ORBALLOC deprecated function 381  
 ORBARGS function 32, 70, 102, 314  
 ORBEXEC function 317  
 ORBFREE deprecated function 381  
 ORBGET deprecated function 381  
 ORBHOST function 323  
 ORBINIT deprecated function 381  
 Orbix E2A COBOL runtime 6, 48, 263  
 Orbix E2A IDL compiler  
   configuration settings 257  
   introduction to 21, 58, 91  
   -M argument 244  
   -O argument 250  
   -Q argument 252  
   running 238  
   -S argument 253  
   specifying arguments for 243  
   -Z argument 256  
 Orbix locator daemon. *See* locator daemon  
 Orbix node daemon. *See* node daemon  
 ORBPUT deprecated function 382  
 ORBREG function 325  
 ORBREGO deprecated function 382  
 ORBREQ deprecated function 382

ORBSRVR function 328

ORBSTAT function 329

ORBTIME function 333

## P

PERSISTENT argument 409  
 plug-ins, introduction to 7  
 pseudo object types in IDL 153

## R

Request Processing policy 409  
 RETAIN argument 409

## S

SEQALLOC function 335  
 SEQDUP function 339  
 SECFREE function 344  
 SEQGET function 347  
 SEQSET function 350  
 sequence type  
   in IDL 152  
   mapping to COBOL 198  
   *See also* memory handling  
 Servant Retention policy 409  
 SERVER\_NAME\_ALREADY\_SET exception 413  
 SERVER\_NAME\_NOT\_SET exception 413  
 server output 46  
 servers  
   building 34, 72, 104  
   introduction to 5  
   running 45  
   writing implementation code for 25, 63, 96  
   writing mainline code for 28, 67, 100  
 SIMPLIDL JCL 238  
 SINGLE\_THREAD\_MODEL argument 410  
 SSL 7  
 STAT\_ALREADY\_CALLED exception 413  
 STRFREE function 355  
 STRGET function 357  
 string type  
   in IDL 142  
   mapping to COBOL 181  
   *See also* memory handling  
 STRLEN function 360  
 STRSET function 362  
 STRSETP function 365  
 STRSETSP deprecated function 382  
 STRTOOBJ function 367

## INDEX

- struct type
  - in IDL 148
  - mapping to COBOL 191
- system exceptions 411

## T

- Thread policy 410
- typedefs, mapping to COBOL 210
- TYPEGET function 369
- type names, mapping to COBOL 165
- TYPESET function 371
- TYPESET\_NOT\_CALLED exception 413

## U

- unbounded sequences, memory handling for 385
- unbounded strings, memory handling for 389
- UNEXPECTED\_INVOCATION exception 412
- union type
  - in IDL 149
  - mapping to COBOL 193
- UNKNOWN exception 412
- UNKNOWN\_OPERATION exception 412
- UNKNOWN\_TYPECODE exception 412
- USE\_ACTIVE\_OBJECT\_MAP\_ONLY argument 409
- user exceptions
  - mapping to COBOL 207
  - memory handling for 402
- USER\_ID argument 408

## V

- valuetypes in IDL 137

## W

- wchar type in IDL 144
- wide string type, mapping to COBOL 186
- WSTRFREE function 373
- WSTRGET function 186, 374
- wstring type in IDL 144
- WSTRLEN function 375
- WSTRSET function 186, 376
- WSTRSETP function 377