COBOL Language Mapping Specification

New Edition: June 1999

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm.*

Contents

Contents

Contents

Preface

0.1 About CORBA Language Mapping Specifications

The CORBA Language Mapping specifications contain language mapping information for the following languages:

- Ada
- C
- C++
- COBOL
- IDL to Java
- Java to IDL
- Smalltalk

Each language is described in a separate stand-alone volume.

0.1.1 Alignment with CORBA

The following table lists each language mapping and the version of CORBA that this language mapping is aligned with.

Language Mapping	Aligned with CORBA version
Ada	CORBA 2.0
C	CORBA 2.1
C++	CORBA 2.3
COBOL	CORBA 2.1

Language Mapping	Aligned with CORBA version
IDL to Java	CORBA 2.3
Java to IDL	CORBA 2.3
Smalltalk	CORBA 2.0

0.2 Definition of CORBA Compliance

The minimum required for a CORBA-compliant system is adherence to the specifications in CORBA Core and one mapping. Each additional language mapping is a separate, optional compliance point. Optional means users aren't required to implement these points if they are unnecessary at their site, but if implemented, they must adhere to the *CORBA* specifications to be called CORBA-compliant. For instance, if a vendor supports C++, their ORB must comply with the OMG IDL to C++ binding specified in this manual.

Interoperability and Interworking are separate compliance points. For detailed information about Interworking compliance, refer to the *Common Object Request Broker: Architecture and Specification, Interworking Architecture* chapter.

As described in the *OMA Guide*, the OMG's Core Object Model consists of a core and components. Likewise, the body of *CORBA* specifications is divided into core and component-like specifications. The structure of this manual reflects that division.

The *CORBA* specifications are divided into these volumes:

1. The *Common Object Request Broker: Architecture and Specification*, which includes the following chapters:
   - **CORBA Core**, as specified in Chapters 1-11
   - **CORBA Interoperability**, as specified in Chapters 12-16
   - **CORBA Interworking**, as specified in Chapters 17-21
2. The Language Mapping Specifications, which are organized into the following stand-alone volumes:
   - **Mapping of OMG IDL to the Ada programming language**
   - **Mapping of OMG IDL to the C programming language**
   - **Mapping of OMG IDL to the C++ programming language**
   - **Mapping of OMG IDL to the COBOL programming language**
   - **Mapping of OMG IDL to the Java programming language**
   - **Mapping of Java programming language to OMG/IDL**
   - **Mapping of OMG IDL to the Smalltalk programming language**

0.3 Acknowledgements

The following companies submitted parts of the specifications that were approved by the Object Management Group to become *CORBA* (including the Language Mapping specifications):

- BNR Europe Ltd.
- Defense Information Systems Agency
- Expersoft Corporation
- FUJITSU LIMITED
- Genesis Development Corporation
- Gensym Corporation
- IBM Corporation
- ICL plc
- Inprise Corporation
- IONA Technologies Ltd.
- Digital Equipment Corporation
- Hewlett-Packard Company
- HyperDesk Corporation
- Micro Focus Limited
- MITRE Corporation
- NCR Corporation
- Novell USG
- Object Design, Inc.
- Objective Interface Systems, Inc.
- OC Systems, Inc.
- Open Group - Open Software Foundation
- Siemens Nixdorf Informationssysteme AG
- Sun Microsystems Inc.
- SunSoft, Inc.
- Sybase, Inc.
- Telefónica Investigación y Desarrollo S.A. Unipersonal
- Visual Edge Software, Ltd.

In addition to the preceding contributors, the OMG would like to acknowledge Mark Linton at Silicon Graphics and Doug Lea at the State University of New York at Oswego for their work on the C++ mapping specification.

0.4 References

The following list of references applies to CORBA and/or the Language Mapping specifications:

IDL Type Extensions RFP, March 1995. OMG TC Document 95-1-35.

The Common Object Request Broker: Architecture and Specification, Revision 2.2, February 1998.

CORBAservices: Common Object Services Specification, Revised Edition, OMG TC Document 95-3-31.

COBOL Language Mapping RFP, December 1995. OMG TC document 95-12-10.

COBOL 85 ANSI X3.23-1985 / ISO 1989-1985.

IEEE Standard for Binary Floating-Point Arithmetic, ANIS/IEEE Std 754-1985.

XDR: External Data Representation Standard, RFC1832, R. Srinivasan, Sun Microsystems, August 1995.

OSF Character and Code Set Registry, OSF DCE SIG RFC 40.1 (Public Version), S. (Martin) O'Donnell, June 1994.

RPC Runtime Support For I18N Characters — Functional Specification, OSF DCE SIG RFC 41.2, M. Romagna, R. Mackey, November 1994.

X/Open System Interface Definitions, Issue 4 Version 2, 1995.

Cobol Language Mapping 1

Note – COBOL Language Mapping specification is aligned with CORBA version 2.1.

The OMG document used to update this chapter was orbos/98-05-02.

Contents

This chapter contains the following sections.

1.1 Overview

This COBOL language mapping provides the ability to access and implement CORBA objects in programs written in the COBOL programming language. The mapping is based on the definition of the ORB in *The Common Object Request Broker: Architecture and Specification*. The mapping specifies how CORBA objects (objects defined by OMG IDL) are mapped to COBOL and how operations of mapped CORBA objects are invoked from COBOL.

This chapter is separated into the following sections:

- The Mapping of OMG IDL to COBOL
- The Dynamic COBOL Mapping
- The Type Specific COBOL Mapping

Part I - The Mapping of OMG IDL to COBOL

1.2 Mapping IDL Types to COBOL

This section describes the mapping of OMG IDL types to COBOL. The syntax used within this section generally conforms to the ANSI 85 COBOL standard, as defined within ANSI X3.23-1985-1995. However, there are some extensions beyond ANSI 85 COBOL (such as the use of COBOL typedefs) that are described, but due to their nature are deemed to be an optional part of the mapping.

1.2.1 Mapping of IDL Identifiers to COBOL

1.2.1.1 Scoped Names

Wherever the COBOL programmer uses a global name for an IDL type, constant, exception, or operation the COBOL global name corresponding to an IDL global name is derived as follows:

- For IDL names being converted into COBOL identifiers or a COBOL literal: convert all occurrences of “::" (except the leading one) into a “-” (a hyphen) and remove any leading hyphens. The “::” used to indicate that global scope will be ignored.

Consider the following example:

```
// IDL
module Sample {
    interface Example {

        short op1();
        long op2();
        ...
    };
};
```

A COBOL group item that defines the argument lists within the Dynamic COBOL mapping would use scoped names, as follows:

```
01    SAMPLE-EXAMPLE-OP1.
          ....
01    SAMPLE-EXAMPLE-OP2.
          ....
```

1.2.1.2 Mapping IDL Identifiers to a COBOL Name

A COBOL name may be up to 30 characters in length and can only consist of a combination of letters, digits, and hyphens. The hyphen may not appear as the first or last character.

Where a COBOL name is to be used, the following steps will be taken to convert an IDL identifier into a format acceptable to COBOL.

1. Replace each underscore with a hyphen.
2. Strip off any leading or trailing hyphens.
3. When an IDL identifier collides with a COBOL reserved word, insert the string "IDL-" before the identifier.
4. If the identifier is greater than 30 characters, then truncate right to 30 characters. If this will result in a duplicate name, truncate back to 27 characters and add a numeric suffix to make it unique.

For example, the IDL identifiers:

```
my_1st_operation_parameter
_another_parameter_
add
a_very_very_long_operation_parameter_number_1
a_very_very_long_operation_parameter_number_2
```

become COBOL identifiers:

```
MY-1ST-OPERATION-PARAMETER
ANOTHER-PARAMETER
IDL-ADD
A-VERY-VERY-LONG-OPERATION-PAR
A-VERY-VERY-LONG-OPERATION-001
```

1.2.1.3 Mapping IDL Identifiers to a COBOL Literal

A COBOL literal is a character string consisting of any allowable character in the character set and is delimited at both ends by quotation marks (either quotes or apostrophes).

Where a COBOL literal is to be used, the IDL identifier can be used directly within the quotes without any truncation being necessary.

1.3 Mapping for Interfaces

1.3.1 Object References

The use of an interface type in IDL denotes an object reference. Each IDL interface shall be mapped directly to an opaque COBOL pointer (or when supported, the COBOL typedef CORBA-Object).

The following example illustrates the COBOL mapping for an interface:

```
interface interface1 {
    long op1(in short parm1);
```

```
};
```

For COBOL interfaces are mapped to an opaque pointer type, as illustrated below:

```
01 INTERFACE1             POINTER.
```

1.3.2 Object References as Arguments

IDL permits specifications in which arguments, return results, or components of constructed types may be object references. Consider the following example:

```
#include “interface1.idl”                // IDL

interface interface2 {
    interface1 op2();
};
```

The above example will result in the following COBOL declaration for the interface:

```
01 INTERFACE2             POINTER.
...
```

The following is a sample of COBOL code that may be used to call **op2** using the Type Specific COBOL mapping.

```
WORKING-STORAGE SECTION.
...
01 INTERFACE1-OBJ         POINTER.
01 INTERFACE2-OBJ         POINTER.
01 EV                     TYPE CORBA-ENVIRONMENT.
...
PROCEDURE DIVISION.
   ...
   CALL  "INTERFACE2-OP2" USING
                    INTERFACE2-OBJ
                    EV
         INTERFACE1-OBJ
   ...
```

1.4 Mapping for Basic Data Types

All the IDL basic data types are mapped to the most appropriate COBOL representation for that IDL type. The following table illustrates this mapping.

Table 1-1 Mapping for Basic Data Types

IDL Name	COBOL Representation	Integer Range	COBOL Typedef
short	PIC S9(5) BINARY	-2^15 to 2^15	CORBA-short
long	PIC S9(10) BINARY	-2^31 to 2^31	CORBA-long
long long	PIC S9(18) BINARY	+/- 18 numerics	CORBA-long-long
unsigned short	PIC S9(05) BINARY	0 to 2^16	CORBA-unsigned-short
unsigned long	PIC S9(10) BINARY	0 to 2^32	CORBA-unsigned-long
unsigned long long	PIC S9(18) BINARY	18 numerics	CORBA-unsigned-long-long
float	COMP-1		CORBA-float
double	COMP-2		CORBA-double
char	PIC X		CORBA-char
wchar	PIC G		CORBA-wchar
boolean	PIC 9		CORBA-boolean
octet	PIC X		CORBA-octet
enum	PIC S9(10) BINARY		CORBA-enum

Note that the use of COBOL typedefs is an optional part of this language mapping.

1.4.1 Basic Integer Types

The Basic IDL Integer data types have specific limits. The ORB will be responsible for ensuring that any values do not exceed the specified integer value ranges. If a value outside the permitted range is detected, the ORB will raise an exception.

The mapping of **long long**, and **unsigned long long** was made to PIC S9(18) and PIC 9(18). This is because these are the highest integer values permitted by ANSI 85 COBOL. If a value greater than 18 numeric digits is detected, the ORB will raise an exception.

1.4.2 Boolean

The COBOL mapping of **boolean** is mapped to a PIC 9(1) COBOL 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

Consider the following example:

```
interface Example {                        //IDL
    boolean my_boot;
    ...
};
```

The above example will result in the following COBOL declarations:

```
01 EXAMPLE-MY-BOOL          PICTURE 9(1).
      88 MY-BOOL-FALSE        VALUE 0.
      88 MY-BOOL-TRUE         VALUE 1.
```

1.4.3 enum

The COBOL mapping of **enum** is an unsigned integer capable of representing 2**32 enumerations. Each identifier in an enum has a COBOL condition defined with the appropriate unsigned integer value conforming to the ordering constraints.

Consider the following example:

```
interface Example {                        // IDL
    enum temp{cold, warm, hot}
    ...
};
```

The above example will result in the following COBOL declarations:

```
01 EXAMPLE-TEMP         PICTURE   9(10) BINARY.
      88 TEMP-COLD         VALUE 0.
      88 TEMP-WARM         VALUE 1.
      88 TEMP-HOT          VALUE 2.
```

COBOL code that would use this simple example is as follows:

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

1.5 Mapping for any Types

The IDL **any** type permits the specification of values that can express any IDL type.

- It is mapped to an opaque type pointed to by a COBOL POINTER.
- The contents of the **any** type cannot be accessed directly.
- The auxiliary functions ANYGET, ANYSET, and ANYFREE are provided to manipulate the **any** data.

- The auxiliary functions TYPEGET and TYPESET are provided to manipulate the type of an **any**.

1.5.1 any Mapping

Consider the following example:

```
interface Example {                    //IDL
    any my_any;
    ...
};
```

The above example will result in the following COBOL declarations:

```
01 EXAMPLE-MY-ANY          USAGE POINTER.
```

1.5.2 any Manipulation

1.5.2.1 Client side any handling

Within clients invoking an interface operation:

- For each IN and INOUT any:
  - TYPESET is first used to specify the type within the any.
  - ANYSET is then used to insert the data into the any.
- For each OUT and RETURN any:
  - No initialization is required.

Within clients receiving the results of an invocation of an interface operation:

- For each IN any:
  - No processing is required, the ORB automatically releases the contents of the any.
- For each INOUT, OUT, and RETURN any:
  - TYPEGET is first used to get the type of the data within the any.
  - ANYGET is then used to get the data from the any.
  - ANYFREE should be used to release the any when it is no longer required.

1.5.2.2 Object implementation any handling

Within object implementations receiving an inbound request from a client:

- For each IN and INOUT:
  - TYPEGET is first used to get the type of the data within the any.
  - ANYGET is then used to get the data from the any.
- For each OUT and RETURN any:
  - No processing is required.

Within object implementations sending a response back to clients:

- For each IN any:
  - No processing is required.
  - Once control has been returned to the ORB, the ORB will release the contents of the any.
- For each INOUT, OUT and RETURN any:
  - TYPESET is first used to specify the type within the any.
  - ANYSET is then used to insert the data into the any.
  - Once control has been returned to the ORB, the contents of the any will be transmitted back to the client and then automatically released by the ORB.

1.6 Mapping for Fixed Types

For COBOL, the IDL **fixed** type is mapped to the native fixed-point decimal type. Consider the following example:

```
Interface example {                         //IDL
    attribute fixed<8,2> salary;
    attribute fixed<4,-6> millions;
    attribute fixed<2, 4> small;
```

The above example will result in the following COBOL declarations:

```
01 EXAMPLE-SALARY    PICTURE S9(06) V9(02) PACKED-DECIMAL.
01 EXAMPLE-MILLIONS  PICTURE S9(04) P(06) PACKED-DECIMAL.
01 EXAMPLE-SMALL     PICTURE VPP99  PACKED-DECIMAL.
```

Note – ANSI 85 COBOL limits numeric data items to a maximum of 18 digits; and the IDL fixed type specifies support for up to 31 digits. If the IDL definition passes a value to COBOL of more than 18 digits, the ORB will raise an exception. Passing data from COBOL to a fixed type with greater than 18 digits results in zero fill of the excess most significant digits.

1.7 Mapping for Struct Types

IDL structures map directly onto COBOL group items. Consider the following example:

```
Interface example {                                 //IDL
    struct test {
        long            member1, member2;
        boolean         member3;
    };
    ...
};
```

The above example will result in the following COBOL declarations:

```
01 EXAMPLE-TEST.
   03 MEMBER1              PICTURE S9 (10) BINARY.
   03 MEMBER2              PICTURE S9 (10) BINARY.
   03 MEMBER3              PICTURE 9.
      88 MEMBER3-FALSE        VALUE 0.
      88 MEMBER3-TRUE         VALUE 1.
```

1.8 Mapping for Union Types

IDL discriminated Unions are mapped onto COBOL group items with the REDEFINES clause. Consider the following example:

```
Interface example {                         //IDL
   union test switch(short){
      case 1:   char      case_1;
      case 2:   double    case_2;
      default   long      default_case;
   } test;

   ...
};
```

The above example will result in the following COBOL declarations:

```
01 EXAMPLE-TEST.
   03 D                       PICTURE S9 (05) BINARY.
   03 U.
         05 CASE-2            COMPUTATIONAL-2.
   03 FILLER REDEFINES U.
         05 DEFAULT-CASE      PICTURE S9(10) BINARY.
   03 FILLER REDEFINES U.
         05 CASE-1            PICTURE X.
```

The union discriminator in the group item is always referred to as **D**. The union items are contained within the group item referred to as **U**. Reference to union elements is done using the EVALUATE statement to test the discriminator.

```
EVALUATE D OF EXAMPLE-TEST
   WHEN 1
      DISPLAY "CHAR VALUE IS" CASE-1 OF EXAMPLE-TEST
   WHEN 2
      DISPLAY "LONG VALUE IS" CASE-2 OF EXAMPLE-TEST
   WHEN OTHER
      DISPLAY "DOUBLE VALUE IS"
               DEFAULT-CASE OF EXAMPLE-TEST
END-EVALUATE
```

Note – The ANSI 85 COBOL REDEFINES clause can only be used to specify a redefinition whose actual storage is either the same size or smaller than the area being redefined. As a result, the **union** elements need to be sorted by size from largest to smallest within the generated COBOL structure (as illustrated within the above example).

1.9 Mapping for Sequence Types

The IDL data type sequence permits passing of bounded and unbounded arrays between objects. The following illustrates a bounded sequence of 8 longs, followed by an unbounded sequence of any number of longs:

```
sequence<long,8>      vec8
sequence>long>        vec
```

In COBOL, bounded and unbounded sequences are represented by a COBOL group item:

- The group item label is <interface-name>-<idl-identifier>.
- It will contain one instance of the type with the label <idl-identifier>.
- It will contain an opaque pointer to the sequence with the label <idl-sequence>-SEQ.

The contents of the sequence type <idl-sequence>-SEQ cannot be accessed directly.

The auxiliary functions SEQALLOC, SEQGET, SEQSET, SEQLEN, SEQMAX, and SEQFREE are provided to manipulate the sequence data within the opaque type.

1.9.1 Sequence Mapping

The preceding IDL sequences would be mapped to the following structures, each of which contain one instance of the type and the opaque sequence itself.

```
01 EXAMPLE-VEC8.
   03 VEC8                 PIC S9(10) BINARY.
   03 VEC8-SEQ             USAGE POINTER.

01 EXAMPLE-VEC.
   03 VEC                  PIC S9(10) BINARY.
   03 VEC-SEQ              USAGE POINTER.
```

1.9.2 Sequence Manipulation

1.9.2.1 Client side sequence handling

Within clients invoking an interface operation:

- For each IN and INOUT sequence:
  - SEQALLOC is first used to initialize the sequence.
  - SEQSET is then used to insert each sequence element in turn.
- For each OUT and RETURN sequence:
  - No initialization is required.

Within clients receiving the results of an invocation of an interface operation:

- For each IN sequence:
  - No processing is required, the ORB will automatically release the contents of the sequence.
- For each INOUT, OUT, and RETURN sequence:
  - SEQLEN is used to get the number of elements in the sequence.
  - SEQGET is then used to get each of the elements in turn.
  - SEQFREE should be used to release the sequence when it is no longer required.

1.9.2.2 *Object implementation sequence handling*

Within object implementations receiving an inbound request from a client:

- For each IN and INOUT sequence:
  - SEQLEN is used to get the number of elements in the sequence.
  - SEQGET is then used to get each of the elements in turn.
- For each OUT and RETURN sequence:
  - No processing is required.

Within object implementations sending a response back to clients:

- For each INOUT, OUT and RETURN sequence:
  - SEQALLOC is first used to initialize the sequence.
  - SEQSET is then used to insert each sequence element in turn.
  - Once control has been returned to the ORB, the contents of the sequence will be transmitted back to the client and then automatically released by the ORB.

1.9.2.3 *Nested Sequences*

The type specified within a sequence may be another sequence.

- Nested sequences will result in an additional opaque sequence type within the sequence group item.
- Each label of a nested opaque sequence will have a -SEQ suffix.
- SEQFREE will release all nested sequences within a sequence.

Consider the following example:

```
Interface example {
    attribute sequence<sequence<sequence<long>>>nest;
};
```

The above example will result in the following COBOL declarations:

```
01 EXAMPLE-NEST.
   03 NEST                  PICTURE S9(10) BINARY.
   03 NEST-SEQ              USAGE POINTER.
   03 NEST-SEQ-SEQ          USAGE POINTER.
   03 NEST-SEQ-SEQ-SEQ      USAGE POINTER.
```

1.10 Mapping for Strings

In IDL, there are two kinds of string data types - bounded strings and unbounded strings:

```
string<8> a_bounded_string
string   an_unbounded_string
```

In COBOL, bounded and unbounded strings are represented differently.

- Bounded strings are represented by a PIC X(n) data item, where *n* is the bounded length of the string.
- Unbounded strings are represented by a pointer.

The auxiliary functions STRGET, STRSET, STRSETP, STRFREE, and STRLEN are provided to manipulate unbounded strings.

1.10.1 Bounded String Mapping

Bounded IDL strings are mapped directly to a COBOL PIC X of the specified IDL length. The ORB will be totally responsible for handling the null byte, as required. Inbound strings will have the null byte automatically stripped off by the ORB and outbound strings will automatically have a null byte appended by the ORB.

Consider the following IDL declarations:

```
Interface example {
    attribute string<10> string_1;
};
```

In COBOL, this is mapped directly to:

```
01   EXAMPLE-STRING-1               PIC X(10).
```

1.10.2 Unbounded String Mapping

An unbounded IDL string is mapped to a pointer that is manipulated using the STRGET and STRPUT auxiliary functions.

Consider the following IDL declarations:

```
Interface example {
    attribute string string_2;
};
```

In COBOL, this is represented as:

```
01   EXAMPLE-STRING-2                    POINTER.
```

1.10.2.1 *Client Side Unbounded String Handling*

Within clients invoking an interface operation:

- For each IN and INOUT unbounded string:
  - STRSET (or STRSETP) is used to create the unbounded string.
- For each OUT and RETURN unbounded string:
  - No initialization is required.

Within clients receiving the results of an invocation of an interface operation:

- For each IN unbounded string:
  - No processing is required, the ORB will automatically release the contents of the unbounded string.
- For each INOUT, OUT, and RETURN unbounded string:
  - STRSET (or STRSETP) is used to create the unbounded string.
  - STRFREE should be used to release the unbounded string when it is no longer required.

1.10.2.2 *Object Implementation Unbounded String Handling*

Within object implementations receiving an inbound required from a client:

- For each IN and INOUT unbounded string:
  - STRGET is used to extract the contents of the unbounded string.
- For each OUT and RETURN unbounded string:
  - No processing is required.

Within object implementations sending a response back to clients:

- For each IN unbounded string:
  - No processing is required.
  - Once control has been returned to the ORB, the contents of the unbounded string will be automatically released.
- For each INOUT, OUT, and RETURN unbounded string:
  - STRSET (or STRSETP) is used to create the unbounded string.
  - Once control has been returned to the ORB, the contents of the unbounded string will be transmitted back to the client and then automatically released by the ORB.

1.10.3 Wstring Mapping

The mapping for wstring is similar to the mapping for string, but requires DBCS support from the COBOL compiler.

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.

1.11 Mapping for Arrays

IDL arrays map to the COBOL OCCURS clause, as follows:

- The top level item will take the name <interface-name>-<idl-identifier>.
- Successive levels going down will be named <idl-interface>-<numeric>.
- The actual item itself will be named <idl-identifier>.

For example, given the following IDL definition:

```
Interface example {
   attribute short ShortArray[2][3][4][5];
};
```

The COBOL mapping will generate the following:

```
01 EXAMPLE-SHORTARRAY.
   03 SHORTARRAY-1              OCCURS 2.
    05 SHORTARRAY-2             OCCURS 3.
      07 SHORTARRAY-3           OCCURS 4.
         09 SHORTARRAY-4        OCCURS 5.
            11 SHORTARRAY       PICTURE S9b(5) BINARY.
```

1.12 Mapping for Exception Types

Each IDL exception type is mapped to the following two COBOL group items:

1. A COBOL group-item containing the layout of all the exception values within the IDL module. Since IDL exceptions are allowed to have no members, but COBOL groups must have at least one item, IDL exceptions with no members map to COBOL groups with one member. This member is opaque to applications. Both the type and the name of this single member are implementation-specific.
2. A COBOL group item containing a unique identifier for the exception. The unique identifier for the exception will be in a string literal form.

1.12.1 Exception Mapping

If we consider the following IDL:

```
interface example {
    exception err {
        long value;
    };
};
```

It would be mapped to the following COBOL group items:

```
01 EXCEPTION-ERR.
   03 VALUE            PIC 9(10) BINARY.

01 EX-EXAMPLE-ERR      PICTURE X(...)
                          VALUE "(UNIQUE EXCEPTION ID)".
```

1.13 Mapping for Attributes

IDL attribute declarations are mapped to a pair of simple accessing operations; one to get the value of the attribute and one to set it.

Both the Dynamic COBOL mapping and the Type Specific COBOL mapping contain specific examples of the mapping for attributes.

1.14 Pseudo Objects

There are no exceptions to the COBOL mapping rules. Pseudo-objects are mapped from the pseudo-IDL according to the normal IDL mapping rules specified for COBOL.

1.15 Auxiliary Datatype Routines

1.15.1 Overview

The following auxiliary functions are provided to enable the manipulation of IDL data types that are opaque within a COBOL context.

- unbounded string auxiliary functions
  - STRGET - extract string value into a PIC X(nn) area
  - STRSET - create string using a PIC X(nn) value
  - STRSETP - create string using a PIC X(nn) value and keep trailing spaces
  - STRLEN - get length of a string
  - STRFREE - release string value memory
- unbounded wstring auxiliary functions
  - WSTRGET - extract wstring value into a PIC G(nn) area
  - WSTRSET - create wstring using a PIC G(nn) value
  - WSTRSETP - create wstring using a PIC G(nn) value and keep trailing spaces
  - WSTRLEN - get length of a wstring

- WSTRFREE - release wstring value memory
- sequence auxiliary functions
  - SEQALLOC - allocates data for a sequence
  - SEQFREE - release sequence value memory
  - SEQGET - extracts a specific element from a sequence
  - SEQLEN - returns number of elements in sequence
  - SEQMAX - returns maximum size of sequence
  - SEQSET - stores a specific element into a sequence
- any auxiliary functions
  - ANYGET - extracts data out of an any
  - ANYSET - inserts data into an any
  - ANYFREE - releases an any
  - TYPEGET - returns type of data in the any
  - TYPESET - sets type of data in any
- object auxiliary functions
  - OBJTOSTR - convert an object reference into a stringified object reference
  - STRTOOBJ - convert a stringified object reference into an object reference
  - OBJDUP - duplicate an object reference
  - OBJREL - release an object reference

The following subsections examine each of the above auxiliary functions in greater detail. They are in alphabetical order. Within each, the IDL notation for describing operations is used as a meta notation for describing the syntax of each auxiliary function.

1.15.2 ANYGET

Summary

Extracts data out of an ANY.

```
ANYGET(IN  any        Opaque-Any-Type,
      OUT <type>      Any-Data)
```

Description

The ANYGET function provides access to the data in an ANY.

- It is the programmer's responsibility to check the type of the any and supply a data buffer large enough to receive the contents of the any.
- The TYPEGET function is used to obtain the type of the ANY prior to calling ANYGET.
- If no type is set in the ANY, no type will be returned.

Example

```
01 EXAMPLE-MY-ANY    POINTER.

01 WS-SHORT          PICTURE 9(05) BINARY.
01 WS-LONG           PICTURE 9(10) BINARY.
   . . .
PROCEDURE DIVISION
   . . .
   CALL "TYPEGET" USING    EXAMPLE-MY-ANY
                           EXAMPLE-TYPE-CODE
   EVALUATE TRUE
      WHEN EXAMPLE-TYPE-SHORT
            CALL "ANYGET" USING     EXAMPLE-MY-ANY
                                    WS-SHORT
            DISPLAY "ANY SHORT IS " WS-SHORT

      WHEN EXAMPLE-TYPE-LONG
            CALL "ANYGET" USING     EXAMPLE-MY-ANY
                                    WS-LONG
            DISPLAY "ANY LONG IS "  WS-LONG

      WHEN OTHER
            DISPLAY "UNSUPPORTED TYPE IN ANY"
END-EVALUATE
```

1.15.3 ANYFREE

Summary

Releases storage within an ANY that is currently being used to hold a value.

```
ANYFREE(IN any    Opaque-Any-Type)
```

Description

When ANYSET is called, it will allocate storage to hold the actual ANY value. This may then be released using a call to ANYFREE.

If the Any type is not currently set, the operation will be ignored.

Example

```
01 EXAMPLE-MY-ANY             POINTER.
. . .

MOVE 12                    TO    WS-SHORT
SET EXAMPLE-TYPE-SHORT     TO    TRUE
CALL "TYPESET" USING EXAMPLE-MY-ANY
                     EXAMPLE-TYPE-CODE-LENGTH
                     EXAMPLE-TYPE-CODE
```

```
CALL "ANYSET"  USING EXAMPLE-MY-ANY
                     WS-SHORT
. . .

CALL "ANYFREE" USING EXAMPLE-MY-ANY
```

1.15.4 ANYSET

Summary

Inserts data into an ANY.

```
ANYSET(IN any          Opaque-Any-Type,
      IN <type>       Any-Data)
```

Description

The ANYSET function stores the supplied data into the ANY.

- Users must first set the type of the ANY using TYPESET before calling ANYSET. If no previous type has been set, a CORBA exception will be raised.
- The storage within the ANY will be allocated by the ANYSET call, and will be owned by the ORB.
- Client side users will be responsible for calling ANYFREE to release an ANY type that they either send or receive once they have finished with it.

Example

```
01 EXAMPLE-MY-ANY                POINTER.
. . .

MOVE 12                    TO    WS-SHORT
SET EXAMPLE-TYPE-SHORT     TO    TRUE
CALL "TYPESET" USING    EXAMPLE-MY-ANY
                        EXAMPLE-TYPE-CODE-LENGTH
                        EXAMPLE-TYPE-CODE
CALL "ANYSET"  USING    EXAMPLE-MY-ANY
                        WS-SHORT
```

1.15.5 OBJDUP

Summary

Duplicates an object reference.

```
OBJDUP(IN  pointer      Object-Reference,
       OUT pointer      Duplicate-Object-Reference)
```

Description

The OBJDUP auxiliary function creates another reference to the same object.

Example

```
01   OBJ-REF                    POINTER.
01   OBJ-DUP-REF                POINTER

PROCEDURE DIVISION.

CALL "OBJDUP" USING OBJ-REF
                      OBJ-DUP-REF
```

1.15.6 OBJREL

Summary

Releases an object reference.

```
OBJREL(IN pointer Object-Reference)
```

Description

The OBJREL auxiliary function disassociates the parameter from any object reference.

Example

```
01   OBJ-REF                   POINTER.

PROCEDURE DIVISION.
   ....
   CALL "OBJREL" USING OBJ-REF
```

1.15.7 OBJTOSTR

Summary

Returns a stringified object reference from an object reference.

```
OBJTOSTR(IN  pointer       Object-Reference,
         OUT pointer       Opaque-String-Type)
```

Description

The OBJTOSTR auxiliary function creates a stringified object reference from a valid object reference.

The returned string is an opaque string that is accessed using the STRGET auxiliary routine.

Example

```
01   OBJ-REF                    POINTER.
01   OBJECT-STRING              POINTER.


PROCEDURE DIVISION.
   ....
   MOVE LENGTH OF OBJECT-STRING  TO  OBJECT-STRING-LEN
   CALL "OBJTOSTR" USING   OBJ-REF
                           OBJECT-STRING
```

1.15.8 SEQALLOC

Summary

Allocates control data for a sequence.

```
SEQALLOC(IN  unsigned long    Initial-Maximum-Count,
         IN  <TYPE>           Sequence-Typecode,
         OUT sequence         Opaque-Sequence)
```

Description

The SEQALLOC auxiliary function initializes the opaque sequence control area.

- The maximum count will be set to the maximum value specified.
- For unbounded sequences, the maximum value should be set to the highest numeric value allowed in the field (ten numeric nines).
- The current length will be set to zero.
- The sequence typecode specifies the type of elements within the sequence.

Example

```
WORKING-STORAGE SECTION.
01 EXAMPLE-VECTOR-SEQUENCE.
   03 VECTOR                    COMP-1.
   03 VECTOR-SEQ                POINTER.

01 SEQ-MAX-LENGTH               PICTURE 9(10) BINARY.
   . . .

PROCEDURE DIVISION.
   . . .
   MOVE 10                TO SEQ-MAX-TYPE
   CALL "SEQALLOC" USING SEQ-MAX-LENGTH
                         TYPE-FLOAT
                         VECTOR-SEQ
```

1.15.9 SEQFREE

Summary

Releases a sequence.

```
SEQFREE(IN sequence     Opaque-Sequence-Type)
```

Description

The SEQFREE auxiliary function releases a sequence.

- SEQFREE releases any types currently within the sequence.
- Nested sequences will also be handled.
- If the opaque sequence type has not been allocated, the SEQFREE will be ignored.

Example

```
WORKING-STORAGE SECTION.
01 EXAMPLE-VECTOR-SEQUENCE.
   03 VECTOR                         COMP-1.
   03 VECTOR-SEQ                     POINTER.

PROCEDURE DIVISION.
   . . .
   CALL "SEQFREE" USING    VECTOR-SEQ
```

1.15.10 SEQGET

Summary

Copies a specific element from a sequence into a data area.

```
SEQGET(IN  sequence        Opaque-Sequence-Type,
       IN  unsigned long   Sequence-Element-Index,
       OUT <type>          Sequence-Element)
```

Description

The SEQGET auxiliary function provides access to a specific element of a sequence.

- The data is copied into the data area.
- If the opaque sequence type has not been allocated, a CORBA exception is raised.
- If the requested element is greater than the current length of the sequence, a CORBA exception is raised.

Example

```
WORKING-STORAGE SECTION.
01 EXAMPLE-VECTOR-SEQUENCE.
   03 VECTOR                      COMP-1.
   03 VECTOR-SEQ                  POINTER.

01 ELEMENT-INDEX                  PICTURE 9(10) BINARY.
01 SEQ-LENGTH                     PICTURE 9(10) BINARY.
      . . .

PROCEDURE DIVISION.
   . . .
   CALL "SEQLEN" USING     VECTOR-SEQ
                           SEQ-LENGTH
   PERFORM VARYING ELEMENT-INDEX FROM 1 BY 1
            UNTIL ELEMENT-INDEX > SEQ-LENGTH
         CALL "SEQGET" USING     VECTOR-SEQ
                                 ELEMENT-INDEX
                                 VECTOR
         PERFORM PROCESS-SEQUENCE-ENTRY
   END-PERFORM
   . . .
```

1.15.11 SEQLEN

Summary

Retrieves the current number of elements within a sequence.

```
SEQLEN(IN   sequence          Opaque-Sequence-Type,
       OUT unsigned long      Number-Of-Sequence-Elements)
```

Description

The SEQLEN auxiliary function returns the current number of elements that are within a sequence.

If the opaque sequence type has not been allocated, a CORBA exception is raised.

Example

```
WORKING-STORAGE SECTION.
01 EXAMPLE-VECTOR-SEQUENCE.
   03 VECTOR                      COMP-1.
   03 VECTOR-SEQ                  POINTER.

01 SEQ-LENGTH                     PICTURE 9(10) BINARY.
   . . .

PROCEDURE DIVISION.
```

```
    . . .
    CALL "SEQLEN" USING     VECTOR-SEQ
                            SEQ-LENGTH
    . . .
```

1.15.12 SEQMAX

Summary

Retrieves the maximum number of elements a sequence is allowed to hold.

```
SEQMAX(IN  sequence         Opaque-Sequence-Type,
       OUT unsigned long    Max-Number-Of-Seq-Elements)
```

Description

The SEQMAX utility function obtains the maximum number of elements that may be stored within a sequence.

- If the opaque sequence type has not been allocated, a CORBA exception is raised.
- If the opaque sequence is unbounded, the maximum integer value permitted in the long is returned.

Example

```
WORKING-STORAGE SECTION.
01 EXAMPLE-VECTOR-SEQUENCE.
   03 VECTOR                   COMP-1.
   03 VECTOR-SEQ               POINTER.

01 SEQ-MAXIMUM                 PICTURE 9(10) BINARY.
   . . .

PROCEDURE DIVISION.
   . . .
   CALL "SEQMAX" USING      VECTOR-SEQ
                            SEQ-MAXIMUM
   . . .
```

1.15.13 SEQSET

Summary

Stores the data into the element number element of an unbounded sequence.

```
SEQSET(IN sequence          Opaque-Sequence-Type,
       IN unsigned long     ELEMENT_NUMBER
       IN <type>            DATA)
```

Description

The SEQSET auxiliary function stores the current contents of the data area into the sequence.

- If the requested element number already exists, it is overwritten.
- If the opaque sequence type has not been allocated, a CORBA exception is raised.
- If the opaque sequence is bounded, and the requested element number is greater than the current maximum size, a CORBA exception is raised.

Example

```
WORKING-STORAGE SECTION.
01 EXAMPLE-VECTOR-SEQUENCE.
   03 VECTOR                    COMP-1.
   03 VECTOR-SEQ                POINTER.

01 ELEMENT-NUM                  PICTURE 9(10) BINARY.
01 SEQ-MAXIMUM                  PICTURE 9(10) BINARY.
. . .

PROCEDURE DIVISION.
   . . .
   CALL "SEQMAX" USING     VECTOR-SEQ
                           SEQ-MAXIMUM
   PERFORM VARYING ELEMENT-NUM FROM 1 BY 1
            UNTIL ELEMENT-NUM > SEQ-MAXIMUM
      PERFORM PROCESS-INIT-SEQUENCE-ENTRY
      CALL "SEQSET" USING     VECTOR-SEQ
                              ELEMENT-NUM
                              VECTOR
   END-PERFORM
   . . .
```

1.15.14 STRFREE

Summary

Releases a string.

```
STRFREE(IN string Opaque-String-Type)
```

Description

The STRFREE auxiliary function releases a string.

If the opaque string type has not been allocated, the STRFREE will be ignored.

Example

```
WORKING-STORAGE SECTION.
01 EXAMPLE-STRING                POINTER.

PROCEDURE DIVISION.
   . . .
   CALL "STRFREE" USING EXAMPLE-STRING
```

1.15.15 STRGET

Summary

Copies the contents of an opaque unbounded string type into a PIC X(n) data item.

```
STRGET(IN  string          Opaque-String-Type,
       IN  unsigned long   Length-Of-Target-Area,
       OUT <PIC X>         Target-Area)
```

Description

This STRGET auxiliary function copies the characters in the opaque unbounded string type to the specified target area.

- If the string does not contain enough characters to exactly fill the target, then it will be space padded.
- NUL characters will never be copied.
- A CORBA exception is raised if the destination is not large enough to store all the string data.
- A CORBA exception is raised if the opaque string is not allocated.

Example

```
01 MY-STRING                  POINTER

01 DEST                       PICTURE X(64).
01 DEST-LEN                   PICTURE 9(10).
   . . .

PROCEDURE DIVISION.
   . . .
   MOVE LENGTH OF DEST        TO DEST-LEN
   CALL "STRGET" USING        MY-STRING, DEST-LEN, DEST
   . . .
```

1.15.16 STRLEN

Summary

Returns the actual length of an unbounded string.

```
STRLEN  (IN  string           Opaque-String-Type,
         OUT unsigned long    Length)
```

Description

The STRLEN auxiliary function returns the number of characters in an unbounded string.

A CORBA exception is raised if the opaque string is not allocated.

Example

```
01 MY-STRING                 POINTER
01 LEN                       PICTURE 9(09) BINARY.
   . . .

PROCEDURE DIVISION.
   . . .
   CALL "STRLEN"        USING MY-STRING, LEN
```

1.15.17 STRSET & STRSETP

Summary

Allocates storage for an unbounded string, sets the pointer to point to it, then sets the value.

```
STRSET  (OUT string      Opaque-String-Type,
     IN  unsigned long   Length-Of-Cobol-Text-Area,
     IN  <PIC X>         Cobol-Text)
STRSETP(OUT string       Opaque-String-Type,
     IN  unsigned long   Length-Of-Cobol-Text-Area,
     IN  <PIC X>         Cobol-Text)
```

Description

The STRSET auxiliary function creates an unbounded string and copies all the characters from the COBOL text area into it.

If the text contains trailing spaces, these will not be copied to the dest string.

The STRSETP version of this function is identical, except it will copy trailing spaces.

Example

```
01 COBOL-TEXT                  PICTURE X(160).
01 COBOL-TEXT-LTH              PICTURE 9(10) BINARY.

01 MY-STRING-TYPE              POINTER
   . . .

PROCEDURE DIVISION.
   . . .
   MOVE "TEXT-VALUE"           TO COBOL-TEXT
   MOVE LENGTH OF COBOL-TEXT   TO COBOL-TEXT-LTH
   CALL "STRSET" USING      MY-STRING-TYPE,
                            COBOL-TEXT-LTH,
                            COBOL-TEXT
```

1.15.18 STRTOOBJ

Summary

Creates an object reference from a stringified object reference.

```
STRTOOBJ(IN  pointer    Opaque-Stringified-Obj-Ref,
         OUT pointer    Object-Reference)
```

Description

The STRTOOBJ auxiliary function creates an object reference from a stringified object reference string.

- The values passed in is an opaque string type that is set up using the STRPUT auxiliary routine.
- If the string cannot be converted, the object reference is set to NULL.

Example

```
01  OBJECT-REF              POINTER.
01  OBJECT-NAME             POINTER.


PROCEDURE DIVISION.
   CALL "STRTOOBJ" USING   OBJECT-NAME
                           OBJECT-REF

   IF  OBJECT-REF = NULL
         DISPLAY "OBJSET CALL FAILED"
         GO TO EXIT-PRG
   END-IF
```

1.15.19 TYPEGET

Summary

Extracts type name out of an ANY.

```
TYPEGET(IN   any                     Opaque-Any-Type,
       OUT <PIC X>                   <interface>-TYPE-CODE)
```

Description

The TYPEGET auxiliary function returns the type code of the ANY.

- A Specific TYPE-CODE text area is generated for each interface within the IDL generated copy file.
- TYPEGET is used to get the type of the ANY so that the correct buffer is passed to the ANYGET function.
- If opaque any type has not been initialized, a CORBA exception will be raised.

Example

```
01 EXAMPLE-MY-ANY            POINTER.
01 WS-SHORT                  PICTURE 9(05) BINARY.
01 WS-LONG                   PICTURE 9(10) BINARY.
   . . .

 PROCEDURE DIVISION
      . . .
   CALL "TYPEGET"    USING EXAMPLE-MY-ANY
                           EXAMPLE-TYPE-CODE

   EVALUATE TRUE
      WHEN EXAMPLE-TYPE-SHORT
         CALL "ANYGET" USING  EXAMPLE-MY-ANY
                              WS-SHORT
         DISPLAY "SHORT FROM ANY IS " WS-SHORT

      WHEN EXAMPLE-TYPE-LONG
         CALL "ANYGET" USING  EXAMPLE-MY-ANY
                              WS-LONG
         DISPLAY "LONG FROM ANY IS " WS-LONG

      WHEN OTHER
         DISPLAY "UNSUPPORTED TYPE IN ANY"
   END-EVALUATE
```

1.15.20 TYPESET

Summary

Sets the type name of an ANY.

```
TYPEGET(INOUT any       Opaque-Any-Type,
       IN <PIC X>       <interface>-TYPE-CODE
```

Description

The TYPESET auxiliary function, initializes the ANY, then sets the type of the ANY to the supplied typecode.

TYPESET must be done prior to calling ANYSET as ANYSET uses the current typecode information to insert the data into the ANY. If no previous TYPESET is done, a CORBA exception will be raised by ANYSET.

Example

```
01 EXAMPLE-MY-ANY          POINTER.
01 WS-SHORT                PICTURE S9(5) BINARY.
   . . .

PROCEDURE DIVISION.
   . . .
   MOVE 12                 TO    WS-SHORT
   SET EXAMPLE-TYPE-SHORT  TO    TRUE
   CALL "TYPESET" USING    EXAMPLE-MY-ANY
                           EXAMPLE-TYPE-CODE

   CALL "ANYSET"  USING    EXAMPLE-MY-ANY
                           WS-SHORT
```

Part II - Dynamic COBOL Mapping

This is the second of the three subsections, which describes the Dynamic COBOL mapping from the following viewpoints:

- Dynamic COBOL Mapping Fundamentals
- Common Auxiliary Functions
- Object Invocation Auxiliary Functions
- The Portable Object Adapter
- COBOL Object Adapter Functions

1.16 Dynamic COBOL Mapping Fundamentals

1.16.1 Overview

The Dynamic COBOL mapping is designed to encapsulate the following CORBA fundamentals:

- Object Invocation from COBOL clients maps to the concepts within the CORBA Dynamic Invocation Interface (DII).
- The COBOL Object Adapter maps to the concepts within the CORBA Dynamic Skeleton Interface (DSI).

1.16.2 Mapping for Interfaces

For the Dynamic COBOL Mapping, each IDL interface will be mapped to one or more COBOL COPY files with the same name as the interface. They will contain all the definitions required by the Dynamic COBOL mapping, and may be used in conjunction with auxiliary routines to enable COBOL applications to become a CORBA Object Implementation and to access other CORBA Object Implementations.

1.16.3 Contents of the IDL Generated COBOL COPY File

The COBOL COPY file generated for each IDL interface will contain:

- A level 01 operation name block used to establish an operation name.
- A level 01 interface description block.
- One level 01 parameter block for each operation within the interface.
- An optional level 01 parameter block that holds all exception definitions.

1.16.3.1 The Operation Name block

The rules for the Operation Name block are as follows:

- It will be a PIC X definition large enough to hold the largest operation name within the interface.
- It will be named using the following format:

```
01 <interface-name>-OPERATION.
```

- The contents may be set using level 88 values for each operation name within the interface, each of which will be named as follows:

```
88 <interface-name>-<operation-name>
            VALUE "ACTUAL-OPERATION-NAME".
```

- Operation names will be specified as is.
- Attribute accessor names will be composed as follows:

```
_GET_<Attribute-Idl-Identifier>
_SET_<Attribute-Idl-Identifier>
```

1.16.3.2 *Interface Description block*

The Rules for the Interface Description Block are as follows:

- The contents are totally opaque to application developers.
- The precise contents are implementation specific.
- It will be named using the following format:

```
01 <interface-name>-INTERFACE.
```

- It is used in conjunction with the ORBREG call within a client or an object implementation prior to any other auxiliary function call for a specific interface. (ORBREG is used to register the start of activity for the specific interface).

1.16.3.3 *Operation Parameter blocks*

The rules for the Operation Parameter Blocks are as follows:

- Each attribute and operation defined within the interface will result in a COBOL level 01 parameter block.
- Each Operation Parameter Block will be named as follows:

```
<interface-name>-<attribute/method-name>-ARGS.
```

- Each sub-item within the group, will be a mapping of the attribute or operation data type to the appropriate local COBOL data type.
- Each sub-item label will be a mapping of the IDL name to a COBOL name.
- Return values will be mapped to a sub-item with the name "RESULT".

1.16.3.4 *Exception block*

The rules for the Exception Block are as follows:

- If any exceptions are defined within the interface, an exception block will be defined.
- Each exception defined within the interface will result in a definition within the exception block.
- The exception Block will be named as follows:

```
<interface-name>-USER-EXCEPTIONS.
```

1.16.3.5 *Interface COPY file Example*

Consider the following IDL:

```
interface example {
    exception err {
        long value;
    };

    read-only attribute short first;  // 1st attribute
    read-only attribute long second;  // 2nd attribute

    // IDL operations

    void set(in short n, in short m, in long value);
    long get(in short n, in short m);
};
```

This would result in the a COBOL COPY file called example being generated as follows:

```
*========================================================
*   OPERATION AND ATTRIBUTE ARGUMENT BLOCKS
*========================================================
*
*  ATTRIBUTE  : READONLY SHORT FIRST
*
 01 EXAMPLE-FIRST-ARGS.
      03 RESULT            PICTURE S9(05) BINARY.

*
*  ATTRIBUTE  : READONLY LONG SECOND
*
 01 EXAMPLE-SECOND-ARGS.
      03 RESULT            PICTURE S9(10) BINARY.

*
*  OPERATION  :  SET
*  PARAMETERS :  IN SHORT N
```

```
*                IN SHORT M
*                IN LONG VALUE
*
 01 EXAMPLE-SET-ARGS.
      03 N                    PICTURE S9(05) BINARY.
      03 M                    PICTURE S9(05) BINARY.
      03 IDL-VALUE            PICTURE S9(10) BINARY.

*
*  OPERATION  :  GET
*  PARAMETERS :  IN SHORT N
*                IN SHORT M
*  RETURNS    :  LONG
*
 01 EXAMPLE-GET-ARGS.
      03 N                    PICTURE S9(05) BINARY.
      03 M                    PICTURE S9(05) BINARY.
      03 RESULT               PICTURE S9(10) BINARY.

*========================================================
*   EXAMPLE-OPERATION
*========================================================
 01   EXAMPLE-OPERATION          PICTURE X(12).
      88 EXAMPLE-GET-FIRST          VALUE "_GET_FIRST".
      88 EXAMPLE-GET-SECOND         VALUE "_GET_SECOND".
      88 EXAMPLE-SET                VALUE "SET".
      88 EXAMPLE-GET                VALUE "GET".

*========================================================
*   EXAMPLE-INTERFACE
*
*   AN OPAQUE STRUCTURE CONTAINING INTERFACE DETAILS.
*   FOR THIS SPECIFIC ILLUSTRATION, IT HAS BEEN
*   GENERATED IN A SEPARATE COPY FILE THAT IS INCLUDED
*   HERE.
*========================================================
   COPY EXAMPLE1.
*========================================================
*   EXAMPLE-USER-EXCEPTIONS
*========================================================
01 EXAMPLE-USER-EXCEPTIONS.
   03 EXCEPTION-ID            POINTER.
   03 D                       PICTURE 9(10) BINARY.
   03 U                       PICTURE X(<MAX DATA SIZE>).
   03 EXCEPTION-ERR REDEFINES U.
         05 VALUE             PICTURE 9(5) BINARY.

01 EX-EXAMPLE-ERR             PICTURE X(...)
                              VALUE "(UNIQUE EXCEPTION ID)".
```

1.16.4 The Global CORBA COPY File

The CORBA COBOL COPY file contains essential data definitions for the Dynamic COBOL Mapping. Users who use the Dynamic COBOL mapping are required to place this copy file into their WORKING STORAGE section within their COBOL application.

The following data areas are defined within the CORBA COBOL COPY file.

1.16.4.1 COA-REQUEST-INFO

The COA-REQUEST-INFO structure is used within Dynamic COBOL Mapping dispatchers to hold information about the current invocation request. Details of how it is populated, and how it should be used is described within the COBOL Object Adapter subsection below.

```
01 COA-REQUEST-INFO.
   03     INTERFACE-NAME     USAGE IS POINTER.
   03     OPERATION-NAME     USAGE IS POINTER.
   03     PRINCIPAL          USAGE IS POINTER.
   03     TARGET             USAGE IS POINTER.
```

The first three data items are unbounded CORBA character strings. The normal auxiliary STRGET routine for accessing unbounded string should be used to extract the text into PIC X(nn) buffers. TARGET is a COBOL object reference.

1.16.4.2 ORB-STATUS-INFORMATION

The ORB-STATUS-INFORMATION structure is used within Dynamic COBOL Mapping clients to hold the status of the last invocation made on either an object or by a local auxiliary function. Its usage is explained in more detail within the Client viewpoint subsection below.

```
01 ORB-STATUS-INFORMATION.
   03 EXCEPTION-NUMBER     PICTURE 9(9) BINARY.
   03 COMPLETION-STATUS    PICTURE 9(4) BINARY.
      88 COMPLETION-STATUS-YES      VALUE 0.
      88 COMPLETION-STATUS-NO       VALUE 1.
      88 COMPLETION-STATUS-MAYBE    VALUE 2.
   03 FILLER               PICTURE X(02).
   03 EXCEPTION-TEXT       USAGE IS POINTER.
```

For successful method invocations the EXCEPTION-NUMBER will be 0 and COMPLETION-STATUS-YES will be true. In all other instances, an appropriate numeric will be set to indicate a specific exception has been raised.

EXCEPTION-TEXT is a pointer to an unbounded string that describes any exception. The STRGET auxiliary routine is used to access the text.

1.16.5 Mapping for Attributes

IDL attribute declarations are mapped to a pair of simple accessing operations; one to get the value of the attribute and one to set it.

To illustrate this, within the context of the Dynamic COBOL Mapping, consider the following specification:

```
interface foo {
    attribute float balance;
};
```

The following code would be used within a Dynamic Mapping COBOL client to get and set the balance attribute that is specified in the IDL above:

```
*
*   GET the Balance
*
    SET   FOO-GET-BALANCE      TO TRUE
    CALL "ORBEXEC"    USING    FOO-OBJ
                               FOO-OPERATION
                               FOO-GET-BALANCE-ARGS
                               FOO-USER-EXCEPTIONS
    DISPLAY BALANCE IN FOO-GET-BALANCE-ARGS

*
*   SET the Balance
*
    MOVE 12.34                 TO BALANCE
                                IN FOO-SET-BALANCE-ARGS
    SET   FOO-SET-BALANCE      TO TRUE
    CALL "ORBEXEC"    USING    FOO-OBJ
                               FOO-OPERATION
                               FOO-SET-BALANCE-ARGS
                               FOO-USER-EXCEPTIONS
```

1.16.6 Mapping for Typedefs and Constants

Within the Dynamic COBOL Mapping, the parameter lists for IDL operations are unrolled back to their basic COBOL types within the IDL generated COBOL copy file for an interface. As part of this process, the IDL constants and IDL typedefs will be used to resolve the operation arguments as part of the unrolling process.

There will be no direct output into the IDL generated COBOL copy file for either IDL Tyedefs or IDL Constants.

1.16.7 Mapping for Exception Types

All exception definitions for an interface are contained within one COBOL group item in the IDL generated COBOL copy file:

- The block is named:

```
<interface>-USER-EXCEPTIONS
```

- It will contain an EXCEPTION-ID string that will hold a textual description of the exception.
- It will contain the ordinal number of the current exception in a field called D.
- When a user exception is raised, this area will be filled in.
- Exceptions are raised using the COAERR auxiliary routine.
- Each exception within the Exception Block is mapped using normal mapping rules for exceptions.

A separate level 01 will also be defined for each exception to specify a unique exception identifier.

To illustrate the above rules, consider the following IDL:

```
interface example {
    exception err {
        long value;
    };
    exception bad {
        short value;
        short code;
        string reason;
    };
};
```

It would be mapped to the following COBOL group items:

```
01 EXAMPLE-USER-EXCEPTIONS.
   03 EXCEPTION-ID      POINTER.
   03 D                 PIC 9(9) BINARY.
      88 D-ERR             VALUE 1.
      88 D-BAD             VALUE 2.
   03 U                 PIC X(<MAX DATA SIZE>).
   03 EXCEPTION-ERR REDEFINES U.
      05 VALUE          PIC 9(10) BINARY.
   03 EXCEPTION-BAD REDEFINES U.
      05 VALUE          PIC 9(5) BINARY.
      05 CODE           PIC 9(5) BINARY.
      05 REASON         POINTER.

01 EX-EXAMPLE-ERR       PICTURE X(...)
                           VALUE "(UNIQUE EXCEPTION ID)".

01 EX-EXAMPLE-BAD       PICTURE X(...)
                           VALUE "(UNIQUE EXCEPTION ID)".
```

Within the <Interface>-USER-EXCEPTIONS area:

- The EXCEPTION-ID is an unbounded string, so is accessed using STRGET.
- The value D will contain the ordinal value of the union element that contains the exception data.

1.17 Common Auxiliary Routines

1.17.1 Overview

The following Dynamic Mapping auxiliary functions are used within either a client invoking an object method, or within a COBOL object implementation:

- ORBREG - Registers a specific interface for use
- ORBSTAT - Registers a status information buffer

Each of the above is described in more detail below.

1.17.2 ORBREG

Summary

Registers an interface.

```
ORBREG( IN <COBOL STRUCT> Cobol-Interface-Description)
```

Description

Before any activity can occur for a specific interface, by either a client invoking its methods, or within an object implementation initializing itself, the ORBREG call must first be made to register the fact that activity for the interface is about to be started.

- The interface description registered by ORBREG is totally opaque and is generated within the COBOL COPY file generated from the IDL.
- The format for the name of the IDL generated interface description is <interface_name>-interface.
- It may be used to register more than one concurrent interface.

Example

```
   COPY EXAMPLE.
   COPY SAMPLE.
   ...

PROCEDURE DIVISION.
   CALL "ORBREG" USING EXAMPLE-INTERFACE
   CALL "ORBREG" USING SAMPLE-INTERFACE
```

1.17.3 ORBSTAT

Summary

Registers a status information block.

```
ORBSTAT( IN <COBOL STRUCT> Status-Description)
```

Description

ORBSTAT is used to register the status information block, ORB-STATUS-INFORMATION so that the status of successive calls is available.

- ORB-STATUS-INFORMATION is defined in the standard CORBA copybook.
- Within it there is an EXCEPTION-NUMBER field that may be tested. When it is zero, then the last auxiliary function call was successful.
- The status of any auxiliary function call is available.
- The ORBSTAT call should be made before any other auxiliary call.
- ORBSTAT is an optional call. No status information will be available if ORBSTAT is not called.
- It is only called once per program.

Example

```
   COPY CORBA.
   COPY EXAMPLE.
   ...

PROCEDURE DIVISION.
   CALL "ORBSTAT"    USING ORB-STATUS-INFORMATION
   CALL "ORBREG"     USING EXAMPLE-INTERFACE
   IF EXCEPTION-NUMBER NOT = 0
      DISPLAY "ORBREG FAILED (" EXCEPTION-NUMBER ")"
   END-IF
```

1.18 Object Invocation

1.18.1 Overview

For a client to invoke an object, it needs to make the following sequence of calls:

- Call ORBSTAT to register the ORB-STATUS-INFORMATION to enable the gathering of status information.
- Call ORBREG to register one or more specific interfaces.

Each of the above calls are discussed in more details within the previous section. Once they have been completed, the ORBEXEC auxiliary function may be used to invoke operations.

The ORBEXEC auxiliary routine is described in more precise detail below.

Note – The use of the ORBREG and ORBEXEC calls are designed to map to the CORBA DII interface.

1.18.2 ORBEXEC

Summary

Invokes an operation on the object.

```
ORBEXEC  IN    pointer        Object-Reference,
         IN    <PIC X>        Operation-Name,
         INOUT <COBOL STRUCT> Operation-Argument-Buffer)
         OUT   <COBOL STRUCT> User-Exception-Block
```

Description

The ORBEXEC auxiliary function allows a COBOL client to invoke operations on the object implementation represented by the supplied object reference.

- The operation-name will always be in a field, within the IDL generated COBOL copy file for each interface, called:

  ```
  <interface-name>-OPERATION
  ```

- The actual value within operation-name is requested by setting a level 88 for the specific operation to true. The naming convention is as follows:

  ```
  <interface-name>-<operation-name>
  ```

- The operation-buffer, which is used to hold the operation's parameters, is generated within the interface's IDL generated COBOL COPY file. Each operation within an interface has its own specific parameter block that is named using the convention:

  ```
  <interface-name>-<operation-name>-ARGS
  ```

- The user-exception-block, which is used to return any user exceptions that are raised, is generated within the interface's IDL generated COBOL COPY file. By convention it is named:

  ```
  <interface-name>-USER-EXCEPTIONS
  ```

Example

```
01 EXAMPLE-OBJ             POINTER.

   COPY CORBA.
   COPY EXAMPLE.

PROCEDURE DIVISION.
   CALL "ORBSTAT"    USING ORB-STATUS-INFORMATION
   CALL "ORBREG"     USING EXAMPLE-INTERFACE
   ....

*  INVOKE "GET" OPERATION
   SET   EXAMPLE-GET       TO    TRUE
   CALL "ORBEXEC"    USING EXAMPLE-OBJ
                           EXAMPLE-OPERATION
                           EXAMPLE-GET-ARGS
                           EXAMPLE-USER-EXCEPTIONS

   IF  EXCEPTION-NUMBER NOT = 0
      DISPLAY "OPERATION FAILED (" EXCEPTION-NUMBER ")"
      GO TO EXIT-PRG
   END-IF
   ....
```

1.19 The COBOL Object Adapter

1.19.1 Overview

The following Object Implementation details are examined in more detail below:

- Initialization - Registering the interfaces that are to be supported.
- The Dispatcher - A single entry point that is called to handle all the interfaces registered during initialization.
- Operation Execution - How each operation obtains, and then returns its parameters.

1.19.2 Object Implementation Initialization

When a server is started, it must make the following sequence of calls:

- Call ORBSTAT to register the ORB-STATUS-INFORMATION to enable the gathering of status information.
- A series of one or more calls to ORBREG to register the specific implementations that the server supports.
- Call COAINIT to complete the initialization for the server. Note that once COAINIT has been called, it will not return until the server is terminating.

The following example illustrates the initialization of the above:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SERVER.

DATA DIVISION.
WORKING-STORAGE SECTION.

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

   COPY CORBA.
   COPY SAMPLE.
   COPY EXAMPLE.

PROCEDURE DIVISION.

INIT.
   CALL "ORBSTAT"    USING     ORB-STATUS-INFORMATION
   CALL "ORBREG"     USING     EXAMPLE-INTERFACE
   CALL "ORBREG"     USING     SAMPLE-INTERFACE
   CALL "COAINIT"    USING     SERVER-NAME
                               SERVER-NAME-LEN
   STOP RUN.
```

1.19.3 Object Implementation Dispatcher

Each Object Implementation is required to support an operation dispatcher:

- The COBOL program that will be the dispatcher will:
  - have its PROGRAM-ID set to DISPATCH (the name of its main entry point), or
  - contain an entry point statement with DISPATCH in it (ENTRY "DISPATCH").
- It will be called once for each incoming operation invocation.
- It will initially obtain the details of the incoming request using the COAREQ function, then using those details, its will perform the requested function.

The following example illustrates this sequence:

```
IDENTIFICATION DIVISION.
   PROGRAM-ID. DISPATCH.

DATA DIVISION.
WORKING-STORAGE SECTION.

   COPY CORBA.
   COPY EXAMPLE.
   ....

PROCEDURE DIVISION.
```

```
CALL "ORBSTAT" USING    ORB-STATUS-INFORMATION.
CALL "ORBREQ"  USING    REQUEST-INFO.

CALL "STRGET"  USING    OPERATION-NAME
                        EXAMPLE-OPERATION-LENGTH
                        EXAMPLE-OPERATION.

EVALUATE TRUE
   WHEN EXAMPLE-SET
      PERFORM DO-EXAMPLE-SET
   WHEN EXAMPLE-GET
      PERFORM DO-EXAMPLE-GET
   ....
END-EVALUATE
```

In the above example, the Object Implementation only supports one interface, so it only uses the operation name to determine what needs to be done. In other cases, where more than one interface is supported, it would also check the INTERFACE-NAME to determine which interface the incoming request is invoking the operation on.

1.19.4 Object Implementation Operations

Each implementation of an interface operation must initially make a COAGET call to obtain all the parameters for the incoming request. The COAGET call will populate the parameter area that was generated within the interface's IDL generated COBOL copy file.

Once the implementation of an interface's operation has completed its processing, it must make a COAPUT call to return all outgoing parameter values back to the caller. The COAPUT call will extract the outgoing parameters from the operation's parameter area within the IDL generated COBOL copy file.

If an operation takes no parameters and has no return value COAGET and COAPUT must still be called.

The following segment of code now illustrates the usage of COAGET and COAPUT:

```
DO-EXAMPLE-SET.
   CALL "ORBGET"  USING EXAMPLE-SET-ARGS
   PERFORM SET-BUSINESS-LOGIC
   CALL "ORBPUT"  USING EXAMPLE-SET-ARGS
```

1.20 COBOL Object Adapter Functions

1.20.1 Overview

The following COBOL Object Adapter functions are used within a COBOL object implementation.

- Object Implementation - Initialization routines

- COAINIT - Completes initialization of a COBOL Object Implementation.
- Object Implementation - Dispatcher routines
  - COAREQ - Obtain details of current incoming request
  - COAGET - Get all the requests incoming parameters
  - COAPUT - Return all the requests outgoing parameters
  - COAERR - Raise a user exception
  - OBJNEW - Creates a new Object Reference

Each of the above is listed alphabetically and described in more detail below.

1.20.2 COAERR

Summary

Raises a user exception.

```
COAERR(IN <USER-EXCEPTION-BUFFER> Exception-Buf)
```

Description

COAERR is used to raise the current user exception that is set within the exception buffer.

- The programmer must first set the appropriate data in the exception buffer before making the call.
- The buffer is generated automatically from IDL within the interface's COBOL COPY file.
- The EXCEPTION-ID and D must be set within the buffer as well as the appropriate user data.

Example

Consider the following IDL:

```
interface foo {
    exception err {
        long value;
    };

    long bar (in short n, out short m)
        raises err;
}
```

The complete COBOL operation parameter buffer looks like:

```
01 FOO-BAR-ARGS.
   03  N     PICTURE S9(05) BINARY.
   03  M     PICTURE S9(05) BINARY.
   03  RESULTPICTURE S9(10) BINARY.
```

The COBOL code to access this parameter list would be as follows:

```
FOO-BAR-IMPLEMENTATION.
   ...
   IF CODE = ERR
      MOVE CODE TO VALUE
      SET D-ERR TO TRUE
      MOVE LENGTH OF EX-FOO-ERR TO WS-LTH
      CALL "STRSET" USING        EXCEPTION-ID,
                                 WS-LTH,
                                 EX-FOO-ERR
      CALL "COAERR" USING FOO-USER-EXCEPTIONS
   ELSE
      CALL "COAPUT" USING FOO-BAR-ARGS.
   END-IF
```

1.20.3 COAGET

Summary

Populates an operation's parameter buffer with IN and INOUT values:

```
COAGET(INOUT <COBOL STRUCT> Cobol-Operation-Parameter-Buf)
```

Description

COAGET copies the incoming operation's argument values into the complete COBOL operation parameter buffer, that is supplied.

This buffer is generated automatically from IDL within the interface's COBOL COPY file.

- Each operation implementation must begin with a call to COAGET and end with a call to COAPUT.
- Only IN and INOUT values in this structure are populated by this call.
- If the operation takes no parameters and has no return value COAGET and COAPUT must still be called.

Example

Consider the following IDL:

```
interface foo {
   long bar (in short n, out short m);
}
```

The complete COBOL operation parameter buffer looks like:

```
01 FOO-BAR-ARGS.
   03  N          PICTURE S9(05)    BINARY.
```

```
    03  M          PICTURE S9(05)    BINARY.
    03  RESULT     PICTURE S9(10)    BINARY.
```

The COBOL code to access this parameter list would be as follows:

```
FOO-BAR-IMPLEMENTATION.
   CALL "COAGET" USING FOO-BAR-ARGS

   DISPLAY  "N = " N
   MOVE     N   TO M
   MOVE     216 TO RESULT

   CALL "COAPUT" USING FOO-BAR-ARGS.
```

This returns the value of *n* back to the client in the *m* argument, and also sends the result back as the literal value 216.

1.20.4 COAINIT

Summary

Initializes a COBOL Object Implementation.

```
COAINIT(IN <PIC X>              Server-ID,
        IN unsigned long        Server-ID-Length)
```

Description

COAINIT is used to notify the ORB that a server is ready to start receiving requests. The server identifier is passed into this call, along with its length.

- Note that the server identifier is case-sensitive.
- If no previous interface has been registered with an ORBREG call, the COAINIT call will raise a CORBA exception.

Example

```
01  SERVER-ID        PIC X(7) VALUE "Example".
01  SERVER-ID-LTH    PIC 9(9) BINARY.
   . . .

PROCEDURE DIVISION.
   . . .
   MOVE LENGTH OF SERVER-ID TO SERVER-ID-LTH
   CALL "COAINIT" USING    SERVER-ID
                           SERVER-ID-LTH
```

1.20.5 COAPUT

Summary

Takes INOUT, OUT and result values from an operation's parameter buffer and returns the values to the caller.

```
COAPUT(INOUT <COBOL STRUCT> Cobol-Operation-Parameter-Buf)
```

Description

COAPUT takes the outgoing argument values from the complete COBOL operation parameter buffer, and returns them to the client that called the operation.

This buffer is generated automatically from IDL within the interface's COBOL COPY file.

Each operation implementation must begin with a call to COAGET and ends with a call to COAPUT.

- Only INOUT, OUT and the special RESULT OUT items are processed by this call.
- The programmer must ensure that all INOUT, OUT and RESULT values are correctly allocated. Failure to do so will result in a CORBA exception being raised.
- If the operation takes no parameters and has no return value, COAGET and COAPUT must still be called passing in a dummy data area.
- If a user exception has been raised, the COAPUT will do nothing.

Example

Consider the following IDL:

```
interface foo {
   long bar (in short n, out short m);
}
```

The complete COBOL operation parameter buffer looks like:

```
01 FOO-BAR-ARGS.
   03  N           PICTURE S9(05) BINARY.
   03  M           PICTURE S9(05) BINARY.
   03  RESULT      PICTURE S9(10) BINARY.
```

The COBOL code to access this parameter list could looks like:

```
FOO-BAR-IMPLEMENTATION.
   CALL "COAGET" USING FOO-BAR-ARGS

   DISPLAY     "N = " N
   MOVE        N   TO M
   MOVE        216 TO RESULT
```

```
CALL "COAPUT" USING FOO-BAR-ARGS.
```

This returns the value of *n* back to the client in the *m* argument, and sends the result back as the literal value 216.

1.20.6 COAREQ

Summary

Obtain details of current inbound request within Implementation dispatcher.

```
COAREQ( IN <COBOL STRUCT> Request-Info)
```

Description

COAREQ is used within Object Implementation dispatchers to obtain the details of the current incoming invocation request. It will populate the following structure, which is defined in the CORBA COPY file, with the details.

```
01 REQUEST-INFO.
   03  INTERFACE-NAME      POINTER.
   03  OPERATION-NAME      POINTER.
   03  PRINCIPAL           POINTER.
   03  TARGET              POINTER.
```

The first three data items are unbounded CORBA character strings. They can be copied into PIC X(n) buffers using the STRGET auxiliary function. The TARGET is a COBOL object reference for this operation invocation.

- COAREQ must be called exactly once per operation invocation.
- COAREQ must be called after a request has been dispatched to a server and before any calls are made to access the parameter values.

Example

```
WORKING-STORAGE SECTION.
   COPY CORBA.
   ...

PROCEDURE DIVISION.

ENTRY "DISPATCH
   CALL "ORBSTAT"    USING    ORB-STATUS-INFORMATION.
   CALL "ORBREQ"     USING    REQUEST-INFO.

   CALL "STRGET"     USING    OPERATION-NAME
                              INTERFACE-OPERATION-LTH
                              INTERFACE-OPERATION.
   ...
```

1.20.7 OBJNEW

Summary

Creates an Object Reference.

```
OBJNEW(IN  <PIC X>   Server-Name,
       IN  <PIC X>   Interface-Name,
...... IN  <PIC X>   Object-Identifier,
       OUT pointer   Object-Reference)
```

Description

The OBJNEW auxiliary function creates a unique object reference. It is specifically designed for use by the Dynamic COBOL mapping.

- The "Server-Name" is a space terminated server identifier specified on the COAINIT function.
- The "Interface-Name" is a space terminated field containing the interface name.
- The "Object-Identifier" is a space terminated identifier for the object being created (for example, an account number for an account object being created by an account factory object).

Example

```
   COPY EXAMPLE.
   COPY CORBA.
   ...
01 OBJ-REF                    POINTER.
01 OBJECT-IDENTIFIER          PICTURE X(25).
01 SERVER-NAME                PICTURE X(12)
                                 VALUE "SERVER".
02 INTERFACE-NAME             PICTURE X(12)
                                 VALUE "EXAMPLE".


PROCEDURE DIVISION.
   ....
   MOVE "<unique value> TO OBJECT-IDENTIFIER
   CALL "OBJNEW" USING        SERVER-NAME
                              INTERFACE-NAME
                              OBJECT-IDENTIFIER
                              OBJECT-REF
```

Part III - Type Specific COBOL Mapping

This section describes the Type Specific COBOL mapping from the following viewpoints:

- Type Specific COBOL Mapping - Fundamentals
- Type Specific COBOL Mapping - Object Invocation
- Type Specific COBOL Mapping - The Portable Object Adapter

The syntax used within this section generally conforms to the ANSI 85 COBOL standard, as defined within ANSI X3.23-1985 / ISO 1989-1985.

1.21 Type Specific COBOL Mapping - Fundamentals

1.21.1 Memory Management

The standard auxiliary functions MEMALLOC and MEMFREE should be used to allocate and free storage for dynamic data types. The following two subsections describe these functions.

1.21.2 MEMALLOC

Summary

Allocates memory.

```
MEMALLOC( IN  unsigned long   Length-Required,
          OUT pointer         Pointer)
```

Description

MEMALLOC is used to allocate memory at runtime from the program heap.

- The length of the memory is specified.
- If the function succeeds in allocating the requested number of bytes, then the pointer is set to point to the start of this memory.
- If the function fails, the pointer will contain the NULL value.

Example

```
01 PTR          POINTER.
01 LEN          PIC 9(10) BINARY VALUE IS 32.
         ...

CALL "MEMALLOC" USING LEN, PTR
```

1.21.3 MEMFREE

Summary

Free memory.

```
MEMFREE(IN pointer                Pointer )
```

Description

MEMFREE is used to release dynamically allocated memory, via a pointer that was originally obtained using MEMALLOC.

Care should be taken not to attempt to de-reference this pointer after freeing it, as this may result in a run-time error.

Example

```
01 PTR                      POINTER.
01 LEN                      PIC 9(10) BINARY VALUE IS 32.

CALL "MEMALLOC" USING LEN, PTR

 . . .

CALL "MEMFREE" USING PTR
```

For further details of these functions refer to their description within the "Mapping of IDL to COBOL " section.

1.21.4 Mapping for Attributes

IDL attribute declarations are mapped to a pair of simple accessing operations; one to get the value of the attribute and one to set it.

To illustrate this, within the context of the Type Specific Mapping, consider the following specification:

```
interface foo {
    attribute float balance;
};
```

The following code would be used within a CORBA COBOL client to get and set the balance attribute that is specified in the IDL above:

```
CALL "FOO--GET-BALANCE" USING
               A-FOO-OBJECT
               A-CORBA-ENVIRONMENT
      BALANCE-FLOAT

CALL "FOO--SET-BALANCE" USING
```

```
A-FOO-OBJECT
BALANCE-FLOAT
A-CORBA-ENVIRONMENT
```

There are two hyphen characters ("--") used to separate the name of the interface from the words "get" or "set" in the names of the functions.

The functions can return standard exceptions but not user-defined ones since the syntax of attribute declarations does not permit them.

1.21.5 Mapping for Typedefs

IDL Typedefs are mapped directly to COBOL Typedefs.

1.21.6 Mapping for Constants

The concept of constants does not exist within pure ANSI 85 COBOL. If the implementors COBOL compiler does not support this concept, then the IDL compiler will be responsible for the propagation of constants.

Constant identifiers can be referenced at any point in the user's code where a literal of that type is legal. In COBOL, these constants may be specified by using the COBOL **>>CONSTANT** syntax.

The syntax is used to define a constant-name, which is a symbolic name representing a constant value assigned to it. The following is an example of this syntax:

```
>>CONSTANT MY-CONST-STRING IS "THIS IS A STRING VALUE".
>>CONSTANT MY-CONST-NUMBER IS 100.
```

1.21.7 Mapping for Exception Types

Each defined exception type is mapped to a COBOL group-item along with a constant name that provides a unique identifier for it. The unique identifier for the exception will be in a string literal form.

For example:

```
exception foo {
   long a_supplied_value;
};
```

will produce the following COBOL declarations:

```
01 <SCOPE>-FOO          IS TYPEDEF.
   03 A-SUPPLIED-VALUE  TYPE CORBA-LONG.

>>CONSTANT EX-FOO IS "<UNIQUE ID FOR EXCEPTION>".
```

1.22 Type Specific COBOL Mapping - Object Invocation

1.22.1 Implicit Arguments to Operations

From the point of view of the COBOL programmer, all operations declared in an IDL interface have implicit parameters in addition to the actual explicitly declared operation specific parameters. These are as follows:

- Each operation has an implicit CORBA-Object input parameter as the first parameter; this designates the object that is to process the request.
- Each operation has an implicit pointer to a CORBA-Environment output parameter that permits the return of exception information. It is placed after any operation specific arguments.
- If an operation in an IDL specification has a context specification, then there is another implicit input parameter which is CORBA-Context. If present, this is placed between the operation specific arguments and the CORBA-Environment parameter.
- ANSI 85 COBOL does not support a RETURNING clause, so any return values will be handled as an out parameter and placed at the end of the argument list after CORBA-Environment.

Given the following IDL declaration of an operation:

```
interface example1
{
    float op1(
        in short arg1,
        in long arg2
    );
};
```

The following COBOL call should be used:

```
CALL "EXAMPLE1-OP1" USING
               A-CORBA-OBJECT
               A-CORBA-SHORT
               A-CORBA-LONG
               A-CORBA-ENVIRONMENT
               A-CORBA-FLOAT
```

1.22.2 Argument passing Considerations

All parameters are passed BY REFERENCE.

1.22.2.1 in parameters

All types are passed directly.

1.22.2.2 inout parameters

bounded and fixed length parameters

All basic types, fixed length structures and unions (regardless of whether they were dynamically allocated or specified within WORKING STORAGE) are passed directly (they do not need to change size in memory).

unbounded and variable length parameters

All types that may have a different size upon return are passed indirectly. Instead of the actual parameter being passed, a pointer to the parameter will be passed.

When there is a type whose length may change in size, some special considerations are required. For example; suppose the user wants to pass in a 10 byte unbounded string as an inout parameter. To do this, the address of a storage area that is initially large enough to hold the 10 characters is passed to the ORB. However, upon completion of the operation, the ORB may find that it has a 20 byte string to pass back to the caller. To enable it to achieve this, the ORB will need to deallocate the area pointed to by the address it received, re-allocate a larger area, then place the larger value into the new larger storage area. This new address will then be passed back to the caller.

For all variable length structures, unions and strings that may change in size:

- The caller must initially dynamically allocate storage using the MEMALLOC function and initialize it directly , or use an appropriate accessor function that will dynamically allocate storage (COBOL-xxx-set, where xxx is the type being set up).
- The pointer to the inout parameter is passed.
- When the call has completed and the user has finished with the returned parameter value, the caller is responsible for de-allocating the storage. This is done by making a call to the "MEMFREE" ORB function with the current address in the POINTER.

1.22.2.3 out and return parameters

Bounded

The caller will initially pass the parameter area into which the out (or return) value is to be placed upon return.

Unbounded

For all sequences, and variable length structures, unions and strings:

- The caller passes a POINTER.
- The ORB will allocate storage for the data type out or return value being returned and then place its address into the pointer.
- The caller is responsible for releasing the returned storage when it is no longer required by using a call to the "MEMFREE" ORB function to deallocate it.

1.22.3 Summary of Argument/Result Passing

The following table is used to illustrate the parameter passing conventions used for **in**, **inout**, **out**, and **return** parameters. Following the table is a key that explains the clauses used within the table.

Table 1-2 Parameter Passing Conventions

Data Type	**in** parameter	**inout** parameter	**out** parameter	**Return** result
short	**<type>**	**<type>**	**<type>**	**<type>**
long	**<type>**	**<type>**	**<type>**	**<type>**
long long	**<type>**	**<type>**	**<type>**	**<type>**
unsigned short	**<type>**	**<type>**	**<type>**	**<type>**
unsigned long	**<type>**	**<type>**	**<type>**	**<type>**
unsigned long long	**<type>**	**<type>**	**<type>**	**<type>**
float	**<type>**	**<type>**	**<type>**	**<type>**
double	**<type>**	**<type>**	**<type>**	**<type>**
long double	**<type>**	**<type>**	**<type>**	**<type>**
boolean	**<type>**	**<type>**	**<type>**	**<type>**
char	**<type>**	**<type>**	**<type>**	**<type>**
wchar	**<type>**	**<type>**	**<type>**	**<type>**
octet	**<type>**	**<type>**	**<type>**	**<type>**
enum	**<type>**	**<type>**	**<type>**	**<type>**
fixed	**<type>**	**<type>**	**<type>**	**<type>**
object	**<type>**	**<type>**	**<type>**	**<type>**
struct (fixed)	**<type>**	**<type>**	**<type>**	**<type>**
struct (variable)	**<type>**	**ptr**	**ptr**	**ptr**
union (fixed)	**<type>**	**<type>**	**<type>**	**<type>**
union (variable)	**<type>**	**ptr**	**ptr**	**ptr**
string (bounded)	**<text>**	**<text>**	**<text>**	**<text>**
string (unbounded)	**<string>**	**<string>**	**<string>**	**<string>**
wstring (bounded)	**<wtext>**	**<wtext>**	**<wtext>**	**<wtext>**
wstring (unbounded)	**<wstring>**	**<wstring>**	**<wstring>**	**<wstring>**
sequence	**<type>**	**ptr**	**ptr**	**ptr**

Table 1-2 Parameter Passing Conventions

array (fixed)	**<type>**	**<type>**	**<type>**	**<type>**
array (variable)	**<type>**	**ptr**	**ptr**	**ptr**
any	**<type>**	**ptr**	**ptr**	**ptr**

Table Key:

Key	**Description**
<type>	Parameter is passed BY REFERENCE
ptr	Pointer to parameter is passed BY REFERENCE For **inout**, the pointer must be initialized prior to the call to point to the data type. For **out** and **return**, the pointer does not have to be initialized before the call and will be passed into the call unintialized. The ORB will then initialize the pointer before control is returned to the caller.
<text>	Fixed length COBOL text (not null terminated)
<string>	Pointer to a variable length NULL terminated string
<wtext>	COBOL wtext (not null terminated)
<wstring>	Pointer to a variable length NULL terminated wstring

1.23 Memory Management

1.23.1 Summary of Parameter Storage Responsibilities

The following table is used to illustrate the storage responsibilities for **in**, **inout**, **out**, and **return** parameters. Following the table is a key that explains the numerics used within the table.

Table 1-3 Parameter Storage Responsibilities

Data Type	**in** parameter	**inout** parameter	**out** parameter	**Return** result
short	**1**	**1**	**1**	**1**
long	**1**	**1**	**1**	**1**
long long	**1**	**1**	**1**	**1**
unsigned short	**1**	**1**	**1**	**1**
unsigned long	**1**	**1**	**1**	**1**

Table 1-3 Parameter Storage Responsibilities

unsigned long long	**1**	**1**	**1**	**1**
float	**1**	**1**	**1**	**1**
double	**1**	**1**	**1**	**1**
long double	**1**	**1**	**1**	**1**
boolean	**1**	**1**	**1**	**1**
char	**1**	**1**	**1**	**1**
wchar	**1**	**1**	**1**	**1**
octet	**1**	**1**	**1**	**1**
enum	**1**	**1**	**1**	**1**
fixed	**1**	**1**	**1**	**1**
object	**2**	**2**	**2**	**2**
struct (fixed)	**1**	**1**	**1**	**1**
struct (variable)	**1**	**3**	**3**	**3**
union (fixed)	**1**	**1**	**1**	**1**
union (variable)	**1**	**3**	**3**	**3**
string (bounded)	**1**	**1**	**1**	**1**
string (unbounded)	**1**	**3**	**3**	**3**
wstring (bounded)	**1**	**1**	**1**	**1**
wstring (unbounded)	**1**	**3**	**3**	**3**
sequence	**1**	**3**	**3**	**3**
array (fixed)	**1**	**1**	**1**	**1**
array (variable)	**1**	**3**	**3**	**3**
any	**1**	**3**	**3**	**3**

Table Key:

Case	**Description**
1	Caller may choose to define data type in WORKING STORAGE or dynamically allocate it. For **inout** parameters, the caller provides the initial value and the callee may change that value (but not the size of the storage area used to hold the value). For **out** and **return** parameters, the caller does not have to initialize it, only provide the storage required. The callee sets the actual value.

2	Caller defines CORBA-Object in WORKING STORAGE or within dynamic storage. For **inout** parameters, the caller passes an initial value. If the ORB wants to reassign the parameter, it will first call "CORBA-Object-release" on the original input value. To continue to use the original object reference passed in as an inout, the caller must first duplicate the object reference by calling "CORBA-Object-duplicate." The client is responsible for the release of ALL specific out and return object references. Release of all object references embedded in other out and return structures is performed automatically as a result of calling "CORBA-free." To explicitly release a specific object reference that is not contained within some other structure, the user should use an explicit call to "CORBA-Object-release."
3	For **inout** parameters, the caller provides a POINTER that points to dynamically allocated storage. The storage is dynamically allocated by a call to "CORBA-alloc." The ORB may deallocate the storage and reallocate a larger/smaller storage area, then return that to the caller. For **out** and **return** parameters, the caller provides an unitialized pointer. The ORB will return the address of dynamically allocated storage containing the out or return value within the pointer. In all cases, the ORB is not allowed to return a null pointer. Also, the caller is always responsible for releasing storage. This is done by using a call to "CORBA-free."

1.24 Handling Exceptions

On every call to an interface operation there are implicit parameters along with the explicit parameters specified by the user. For further details, refer to Section 1.22.2, "Argument passing Considerations," on page 1-53. One of the implicit parameters is the **`CORBA-Environment`** parameter which is used to pass back exception information to the caller.

1.24.1 Passing Exception details back to the caller

The **`CORBA-Environment`** type is partially opaque. The COBOL declaration will contain at least the following:

```
01 CORBA-EXCEPTION-TYPE    IS TYPEDEF  TYPE CORBA-ENUM.
   88 CORBA-NO-EXCEPTION         VALUE 0.
   88 CORBA-USER-EXCEPTION       VALUE 1.
   88 CORBA-SYSTEM-EXCEPTION     VALUE 2.

01 CORBA-ENVIRONMENT       IS TYPEDEF.
   03 MAJOR                TYPE CORBA-EXCEPTION-TYPE.
   ...
```

When a user has returned from a call to an object, the **major** field within the call's **environment** parameter will have been set to indicate whether the call completed successfully or not. It will be set to one of the valid types permitted within the field **CORBA-no-exception**, **CORBA-user-exception**, or **CORBA-system-exception**. If the value is one of the last two, then any exception parameters signalled by the object can be accessed.

1.24.2 Exception Handling Functions

The following functions are defined for handling exception information within the **CORBA-Environment** structure.

1.24.2.1 CORBA-exception-set

CORBA-exception-set allows a method implementation to raise an exception. The **a-CORBA-environment** parameter is the environment parameter passed into the method. The caller must supply a value for the exception-type parameter.

```
CALL "CORBA-EXCEPTION-SET" USING
               A-CORBA-ENVIRONMENT
               A-CORBA-EXCEPTION-TYPE
               A-CORBA-REPOS-ID-STRING
               A-PARAM
```

The value of the exception-type parameter constrains the other parameters in the call as follows:

- If the parameter has the value **CORBA-NO-EXCEPTION**, this is a normal outcome to the operation. In this case, both **repos-id-string** and **param** must be NULL. Note that it is *not* necessary to invoke **CORBA-exception-set** to indicate a normal outcome; it is the default behavior if the method simply returns.
- For any other value, it specifies either a user-defined or system exception. The **repos_id** parameter is the repository ID representing the exception type. If the exception is declared to have members, the **param** parameter must be the exception group item containing the parameters according to the COBOL language mapping. If the exception takes no parameters, **param** must be NULL.

If the **CORBA-Environment** argument to **CORBA-exception-set** already has an exception set in it, that exception is properly freed before the new exception information is set.

1.24.2.2 CORBA-exception-id

CORBA-exception-id returns a pointer to the character string identifying the exception. The character string contains the repository ID for the exception. If invoked on an **environment** that identifies a non-exception, a NULL pointer is returned. Note that ownership of the returned pointer does not transfer to the caller; instead, the pointer remains valid until **CORBA-exception-free()** is called.

```
CALL "CORBA-EXCEPTION-ID" USING
               A-CORBA-ENVIRONMENT
      A-POINTER
```

1.24.2.3 CORBA-exception-value

CORBA-exception-value returns a pointer to the structure corresponding to this exception. If invoked on an **environment** which identifies a non-exception, a NULL pointer is returned. Note that ownership of the returned pointer does not transfer to the caller; instead, the pointer remains valid until **CORBA-exception-free()** is called.

```
CALL "CORBA-EXCEPTION-VALUE" USING
            A-CORBA-ENVIRONMENT
      A-POINTER
```

1.24.2.4 CORBA-exception-free

CORBA-exception-free returns any storage that was allocated in the construction of the **environment** exception. It is permissible to invoke this regardless of the value of the IDL-major field.

```
CALL "CORBA-EXCEPTION-FREE" USING
            A-CORBA-ENVIRONMENT
```

1.24.2.5 CORBA-exception-as-any

CORBA-exception-as-any() returns a pointer to a **CORBA-any** containing the exception. This allows a COBOL application to deal with exceptions for which it has no static (compile-time) information. If invoked on a **CORBA-Environment** which identifies a non-exception, a null pointer is returned. Note that ownership of the returned pointer does not transfer to the caller; instead, the pointer remains valid until **CORBA-exception-free()** is called.

```
CALL "CORBA-EXCEPTION-AS-ANY" USING
            A-CORBA-ENVIRONMENT
            A-CORBA-ANY-RTN
```

1.24.3 Example of How to Handle the CORBA-Exception Parameter

The following example is a segment of a COBOL application that illustrates how the Environment functions described above may be used within a COBOL application to handle an exception.

For the following IDL definition:

```
interface MyInterface {
    exception example1{long reason, ...};
    exception example2(...);

    void MyOperation(long argument1)
        raises(example1, example2, ...);
    ...
}
```

The following would be generated:

```
01 MYINTERFACE             IS TYPEDEF  TYPE CORBA-OBJECT.

01 MYINTERFACE-EXAMPLE1    IS TYPEDEF.
   03 REASON   TYPE CORBA-LONG
   03 ...
>>CONSTANT EX-EXAMPLE1  IS "<UNIQUE EXAMPLE1 IDENTIFIER>".

01 MYINTERFACE-EXAMPLE2    IS TYPEDEF.
   03 ...
>>CONSTANT EX-EXAMPLE2  IS "<UNIQUE EXAMPLE2 IDENTIFIER>".
```

The following code checks for exceptions and handles them.

```
WORKING-STORAGE SECTION.
01 MYINTERFACE-OBJECT    TYPE MYINTERFACE
01 EV                    TYPE CORBA-ENVIRONMENT.
01 ARGUMENT1             TYPE CORBA-LONG
01 WS-EXCEPTION-PTR      POINTER.

01 WS-EXAMPLE1-PTR       POINTER.
   ...

LINKAGE SECTION.
01 LS-EXCEPTION          TYPE CORBA-EXCEPTION-ID.
01 LS-EXAMPLE1           TYPE MYINTERFACE-EXAMPLE1.
   ...

PROCEDURE DIVISION.
   ...
   CALL "MYINTERFACE-MYOPERATION" USING
                         MYINTERFACE-OBJECT
                         ARGUMENT1
```

```
                          EV
    EVALUATE MAJOR IN EV
    WHEN CORBA-NO-EXCEPTION
       CONTINUE

    WHEN CORBA-USER-EXCEPTION
       CALL "CORBA-EXCEPTION-ID" USING EV
              WS-EXCEPTION-PTR

       SET ADDRESS OF LS-EXCEPTION
                    TO   WS-EXCEPTION-PTR
       EVALUATE LS-EXCEPTION
          WHEN EX-EXAMPLE1
           CALL "CORBA-EXCEPTION-VALUE" USING EV
                   WS-EXAMPLE1-PTR
           SET ADDRESS OF LS-EXAMPLE1
                         TO WS-EXAMPLE1-PTR
           DISPLAY "XXXX CALL FAILED : "
                   "EXAMPLE1 EXCEPTION RAISED - "
                   "REASON CODE = "
                   REASON IN LS-EXAMPLE1

          WHEN EX-EXAMPLE2
             ....

       END-EVALUATE
       CALL "CORBA-EXCEPTION-FREE" USING EV

    WHEN CORBA-SYSTEM-EXCEPTION
       ...
       CALL "CORBA-EXCEPTION-FREE" USING EV

 END-EVALUATE
 CALL "CORBA-EXCEPTION-FREE" USING EV
```

1.25 Type Specific COBOL Server Mapping

This section describes the details of the OMG IDL-to-COBOL language mapping that apply specifically to the Portable Object Adapter, such as how the implementation methods are connected to the skeleton.

1.25.1 Operation-specific Details

This section defines most of the details of binding methods to skeletons, naming of parameter types, and parameter passing conventions. Generally, for those parameters that are operation-specific, the method implementing the operation appears to receive the same values that would be passed to the stubs.

1.25.2 PortableServer Functions

Objects registered with POAs use sequences of octet, specifically the PortableServer::POA::ObjectId type, as object identifiers. However, because COBOL programmers will often want to use strings as object identifiers, the COBOL mapping provides several conversion functions that convert strings to ObjectId and vice-versa.

```
CALL "PORTABLESERVER-OBJECTID-TO-STR" USING
         A-PORTABLESERVER-OBJECTID
         A-CORBA-ENVIRONMENT
       A-CORBA-STRING-RTN
....

CALL "PORTABLESERVER-OBJECTID-TO-WST" USING
         A-PORTABLESERVER-OBJECTID
         A-CORBA-ENVIRONMENT
       A-CORBA-WSTRING-RTN
....

CALL "PORTABLESERVER-STR-TO-OBJECTID" USING
         A-CORBA-STRING
         A-CORBA-ENVIRONMENT
   A-PORTABLESERVER-OBJECTID-RTN
....

CALL "PORTABLESERVER-WST-TO-OBJECTID" USING
         A-CORBA-WSTRING
         A-CORBA-ENVIRONMENT
   A-PORTABLESERVER-OBJECTID-RTN
....
```

These functions follow the normal COBOL mapping rules for parameter passing and memory management.

If conversion of an ObjectId to a string would result in illegal characters in the string (such as a NUL), the first two functions raise the CORBA-BAD-PARAM exception.

1.25.3 Mapping for PortableServer::ServantManager::Cookie

Since **PortableServer::ServantManager::Cookie** is an IDL native type, its type must be specified by each language mapping. In COBOL, Cookie maps to pointer.

```
01 COOKIE IS TYPEDEF          USAGE POINTER
```

For the COBOL mapping of the **PortableServer::ServantLocator::preinvoke()** operation, the Cookie parameter maps to a pointer to a Cookie, while for the postinvoke() operation, it is passed as a Cookie:

```
CALL "PORTABLESRV-SERVLOC-PREINVOKE" USING
         A-PORTABLESERVER-OBJECTID-
         A-PORTABLESERVER-POA
```

```
         A-CORBA-IDENTIFIER
         A-COOKIE
...
CALL "PORTABLESRV-SERVLOC-POSTINVOKE" USING
         A-PORTABLESERVER-OBJECTID
         A-PORTABLESERVER-POA
         A-CORBA-IDENTIFIER
         A-COOKIE
         A-PORTABLESERVER-SERVANT
```

1.25.4 Servant Mapping

A servant is a language-specific entity that can incarnate a CORBA object. In COBOL, a servant is composed of a data structure that holds the state of the object along with a collection of method functions that manipulate that state in order to implement the CORBA object.

The PortableServer::Servant type maps into COBOL as follows:

```
01 PORTABLESERVER-SERVANT IS TYPEDEF USAGE POINTER
```

Associated with a servant is a table of pointers to method functions. This table is called an entry point vector, or EPV. The EPV has the same name as the servant type with "-epv" appended . The EPV for PortableServer-Servant is defined as follows:

```
01 PORTABLESERVER-SERVANTBASE-EPV IS TYPEDEF.
   03 PRIVATE         USAGE POINTER.
   03 FINALIZE        USAGE PROCEDURE-POINTER.
   03 DEFAULT-POA     USAGE PROCEDURE-POINTER.

* THE SIGNATURES FOR THE FUNCTIONS ARE AS FOLLOWS
      CALL "FINALIZE" USING
            A-PORTABLESERVER-SERVANT
            A-CORBA-ENVIRONMENT

      CALL "DEFAULT-POA" USING
            A-PORTABLESERVER-SERVANT
            A-CORBA-ENVIRONMENT
       A-PORTABLESERVER-POA
```

The PortableServer-ServantBase-epv "private" member, which is opaque to applications, is provided to allow ORB implementations to associate data with each ServantBase EPV. Since it is expected that EPVs will be shared among multiple servants, this member is not suitable for per-servant data. The second member is a pointer to the finalization function for the servant, which is invoked when the servant is etherealized. The other function pointers correspond to the usual Servant operations.

The actual PortableServer-ServantBase structure combines an EPV with per-servant data, as shown below:

```
*    (VEPV IS A POINTER TO THE EPV)
01 PORTABLESERVER-SERVANTBASE-VEPV IS TYPEDEF POINTER.


01 PORTABLESERVER-SERVANTBASE IS TYPEDEF.
   03 PRIVATE         USAGE POINTER.
   03 VEPV            TYPE PORTABLESERVER-SERVANTBASE-VEPV.
```

The first member is a pointer that points to data specific to each ORB implementation. This member, which allows ORB implementations to keep per-servant data, is opaque to applications. The second member is a pointer to a pointer to a PortableServer-ServantBase-epv. The reason for the double level of indirection is that servants for derived classes contain multiple EPV pointers, one for each base interface as well as one for the interface itself. (This is explained further in the next section). The name of the second member, "vepv," is standardized to allow portable access through it.

1.25.5 Interface Skeletons

All COBOL skeletons for IDL interfaces have essentially the same structure as ServantBase, with the exception that the second member has a type that allows access to all EPVs for the servant, including those for base interfaces as well as for the most-derived interface.

For example, consider the following IDL interface:

```
// IDL
interface Counter {
   long add(in long val);
};
```

The servant skeleton generated by the IDL compiler for this interface appears as follows (the type of the second member is defined further below):

```
01 POA-COUNTER IS TYPEDEF.
   03 PRIVATE         USAGE POINTER.
   03 VEPV            TYPE POA-COUNTER-VEPV.
```

As with PortableServer-ServantBase, the name of the second member is standardized to "vepv" for portability.

The EPV generated for the skeleton is a bit more interesting. For the Counter interface defined above, it appears as follows:

```
01 POA-COUNTER-EPV IS TYPEDEF.
   03 PRIVATE         USAGE POINTER.
   03 ADD             USAGE PROCEDURE-POINTER.
```

Since all servants are effectively derived from PortableServer-ServantBase, the complete set of entry points has to include EPVs for both PortableServer-ServantBase and for Counter itself:

```
01 POA-COUNTER-VEPV IS TYPEDEF.
   03 BASE-EPV         USAGE POINTER.
   03 COUNTER-EPV      USAGE POINTER.
```

The first member of the POA-Counter-vepv struct is a pointer to the PortableServer-ServantBase EPV. To ensure portability of initialization and access code, this member is always named "base-epv." It must always be the first member. The second member is a pointer to a POA-Counter-epv.

The pointers to EPVs in the VEPV structure are in the order that the IDL interfaces appear in a top-to-bottom left-to-right traversal of the inheritance hierarchy of the most-derived interface. The base of this hierarchy, as far as servants are concerned, is always PortableServer-ServantBase. For example, consider the following complicated interface hierarchy:

```
// IDL
interface A {};
interface B : A {};
interface C : B {};
interface D : B {};
interface E : B, C {};
interface F {};
interface G : E, F {
   void foo();
};
```

The VEPV structure for interface G shall be generated as follows:

```
* COBOL
01 POA-G-EPV IS TYPEDEF.
   03 PRIVATE      USAGE POINTER.
   03 FOO          USAGE PROCEDURE-POINTER.

01 POA-G-VEPV IS TYPEDEF.
   03 BASE-EPV     USAGE POINTER.
   03 A-EPV        USAGE POINTER.
   03 B-EPV        USAGE POINTER.
   03 C-EPV        USAGE POINTER.
   03 D-EPV        USAGE POINTER.
   03 E-EPV        USAGE POINTER.
   03 F-EPV        USAGE POINTER.
   03 G-EPV        USAGE POINTER.
```

Note that each member other than the "base-epv" member is named by appending "-epv" to the interface name whose EPV the member points to. These names are standardized to allow for portable access to these items.

1.25.6 Servant Structure Initialization

Each servant requires initialization and etherealization, or finalization, functions. For PortableServer-ServantBase, the ORB implementation shall provide the following functions:

```
CALL "PORTABLESERVER-SERVANTBASEINIT" USING
            PORTABLESERVER-SERVANT
            CORBA-ENVIRONMENT

CALL "PORTABLESERVER-SERVANTBASEFINI" USING
            PORTABLESERVER-SERVANT
            CORBA-ENVIRONMENT
```

These functions are named by appending "Init" and "Fini" to the name of the servant, respectively.

The first argument to the init function shall be a valid PortableServer-Servant whose "vepv" member has already been initialized to point to a VEPV structure. The init function shall perform ORB-specific initialization of the PortableServer-ServantBase, and shall initialize the "finalize" struct member of the pointed-to PortableServer-ServantBase-epv to point to the PortableServer-ServantBaseFini() function if the "finalize" member is NULL. If the "finalize" member is not NULL, it is presumed that it has already been correctly initialized by the application, and is thus not modified. Similarly, if the default-POA member of the PortableServer-ServantBase-epv structure is NULL when the init function is called, its value is set to point to the default-POA function, which returns an object reference to the root POA.

If a servant pointed to by the PortableServer-Servant passed to an init function has a NULL "vepv" member, or if the PortableServer-Servant argument itself is NULL, no initialization of the servant is performed, and the CORBA::BAD_PARAM standard exception is raised via the CORBA-Environment parameter. This also applies to interface-specific init functions, which are described below.

The Fini function only cleans up ORB-specific private data. It is the default finalization function for servants. It does not make any assumptions about where the servant is allocated, such as assuming that the servant is heap-allocated and trying to call MEMFREE on it. Applications are allowed to "override" the fini function for a given servant by initializing the PortableServer-ServantBase-epv "finalize" pointer with a pointer to a finalization function made specifically for that servant; however, any such overriding function must always ensure that the PortableServer-ServantBaseFini function is invoked for that servant as part of its implementation. The results of a finalization function failing to invoke PortableServer-ServantBaseFini are implementation-specific, but may include memory leaks or faults that could crash the application.

If a servant passed to a fini function has a NULL "epv" member, or if the PortableServer-Servant argument itself is NULL, no finalization of the servant is performed, and the CORBA::BAD_PARAM standard exception is raised via the CORBA-Environment parameter. This also applies to interface-specific fini functions, which are described below.

Normally, the PortableServer-ServantBaseInit and PortableServer-ServantBaseFini functions are not invoked directly by applications, but rather by interface-specific initialization and finalization functions generated by an IDL compiler. For example, the init and fini functions generated for the Counter skeleton are defined as follows:

```
IDENTIFICATION DIVISION.
   PROGRAM ID. POA-COUNTER-INIT.
   ...
PROCEDURE DIVISION USING
            A-POA-COUNTER
            A-CORBA-ENVIRONMENT

*
* FIRST CALL IMMEDIATE BASE INTERFACE INIT
* FUNCTIONS IN THE LEFT-TO-RIGHT ORDER OF
* INHERITANCE
*
      CALL "PORTABLESERVER-SERVANTBASEINIT" USING
                  A-POA-COUNTER
                  A-CORBA-ENVIRONMENT

*
* NOW PERFORM POA_COUNTER INITIALIZATION
*
   ...
 END-PROGRAM.


IDENTIFICATION DIVISION.
PROGRAM ID. POA-COUNTER-FINI.
...
PROCEDURE DIVISION USING
               A-POA-COUNTER
               A-CORBA-ENVIRONMENT

*
* FIRST PERFORM POA_COUNTER CLEANUP
*
      ...
*
* THEN CALL IMMEDIATE BASE INTERFACE FINI
* FUNCTIONS IN THE RIGHT-TO-LEFT ORDER OF
* INHERITANCE
*
      CALL "PORTABLESERVER-SERVANTBASEFINI" USING
               A-POA-COUNTER
               A-CORBA-ENVIRONMENT
END-PROGRAM.
```

The address of a servant shall be passed to the init function before the servant is allowed to be activated or registered with the POA in any way. The results of failing to properly initialize a servant via the appropriate init function before registering it or allowing it to be activated are implementation-specific, but could include memory access violations that could crash the application.

1.25.7 Application Servants

It is expected that applications will create their own servant structures so that they can add their own servant-specific data members to store object state. For the Counter example shown above, an application servant would probably have a data member used to store the counter value:

```
01 APPSERVANT IS TYPEDEF.
   03 BASE          TYPE PAO-COUNTER.
   03 VALUE         TYPE CORBA-LONG.
```

The application might contain the following implementation of the **`Counter::add`** operation:

```
IDENTIFICATION DIVISION.
   PROGRAM ID. APP-SERVANT-ADD.
...
LINKAGE SECTION.
01 A-APPSERVANTTYPE APPSERVANT.
...
PROCEDURE DIVISION USING
               A-APPSERVANT
               A-CORBA-LONG
               A-CORBA-ENV
             A-CORBA-LONG-RTN
   ADD A-CORBA-LONG TO VALUE IN A-APPSERVANT
   MOVE VALUE IN A-APPSERVANT TO A-CORBA-LONG-RTN
   EXIT PROGRAM
```

The application could initialize the servant dynamically as follows:

```
WORKING-STORAGE SECTION.
01 BASE-EPV         TYPE PORTABLESERVER-SERVANTBASE-EPV.
01 COUNTER-EPV      TYPE POA-COUNTER-EPV.
01 COUNTER-VEPV     TYPE POA-COUNTER-VEPV.
01 MY-BASE          TYPE POA-COUNTER.
01 MY-SERVANT       TYPE APPSERVANT.
         ...
*  INITIALIZE BASE-EPV
      SET PRIVATE IN BASE-EPV    TO NULL
      SET FINALIZE IN BASE-EPV   TO NULL
      SET DEFAULT-POA IN BASE-EPV
                                 TO ENTRY "MY-DEFAULT-POA"
         ...
```

```
*   INITIALIZE COUNTER-EPV
       SET PRIVATE IN COUNTER-EPV TO NULL
       SET ADD IN COUNTER-EPV
                                TO ENTRY "APP-SERVANT-ADD"
          ...
*   INITIALIZE COUNTER-VEPV
       SET BASE-EPV IN COUNTER-VEPV
                                TO ADDRESS OF BASE-EPV
       SET COUNTER-EPV IN COUNTER-VEPV
                                TO ADDRESS OF COUNTER-EPV
          ...
*   INITIALIZE MY-BASE
       SET PRIVATE IN MY-BASE   TO NULL
       SET VEPV IN MY-BASE
                                TO ADDRESS OF COUNTER-VEPV
          ...
*   INITIALIZE MY-SERVANT
       SET BASE IN MY-SERVANT
                                TO ADDRESS OF MY-BASE
       SET VALUE IN MY-SERVANT  TO 0
```

Before registering or activating this servant, the application shall call:

```
       CALL "POA-COUNTER-INIT" USING
                              MY-SERVANT
                              A-CORBA-ENVIRONMENT
```

If the application requires a special destruction function for my-servant, it shall set the value of the PortableServer-ServantBase-epv "finalize" member either before or after calling POA-Counter-init():

```
       SET FINALIZE IN BASE-EPV
                        TO ENTRY "MY-FINALIZER-FUNC"
```

Note that if the application statically initialized the "finalize" member before calling the servant initialization function, explicit assignment to the "finalize" member as shown here is not necessary, since the PortableServer-ServantBaseInit() function will not modify it if it is non-NULL.

1.25.8 Method Signatures

With the POA, implementation methods have signatures that are identical to the stubs except for the first argument. If the following interface is defined in OMG IDL:

```
// IDL
interface example4 {
        long op5(in long arg6);
};
```

A COBOL program for the op5 operation must have the following signature:

```
IDENTIFICATION DIVISION.
      PROGRAM ID. OP5.
      ...
PROCEDURE DIVISION USING
                  SERVANT
                  ARG6
                  ENV
          RTN
   ...
```

The Servant parameter (which is an instance of PortableServer-Servant) is the servant incarnating the CORBA object on which the request was invoked. The method can obtain the object reference for the target CORBA object by using the POA-Current object. The **env** parameter is used for raising exceptions. Note that the names of the **servant** and **env** parameters are standardized to allow the bodies of method functions to refer to them portably.

The method terminates successfully by executing an EXIT PROGRAM statement after setting the declared operation return value. Prior to returning the result of a successful invocation, the method code must assign legal values to all out and inout parameters.

The method terminates with an error by executing the CORBA-exception-set operation (described in Section 1.24.2, "Exception Handling Functions," on page 1-59) prior to executing an EXIT PROGRAM statement. When raising an exception, the method code is not required to assign legal values to any out or inout parameters. Due to restrictions in ANSI85 COBOL, it must return a legal function value.

1.25.9 Mapping of the Dynamic Skeleton Interface to COBOL

The Dynamic Skeleton Interface chapter of the *CORBA* specification contains general information about the Dynamic Skeleton Interface (DSI), and its mapping to programming languages. Within this section, the following topics are covered:

- Mapping the ServerRequest Pseudo Object to COBOL
- Mapping the DynamicImplementationRoutine to COBOL

1.25.9.1 Mapping of the ServerRequest to COBOL

The pseudo IDL for the Dynamic Skeleton Interface's ServerRequest is as follows:

```
module CORBA {
    interface ServerRequest {
        Identifier      operation();
        Context         ctx();
        void            arguments(inout NVList parms);
        void            set_result(any value);
        void            set_exception(
                            exception_type major,
                            any value
                              );
```

```
        }
}
```

The above ServerRequest pseudo IDL is mapped to COBOL as follows:

1.25.9.2 operation

This function returns the name of the operation being performed, as shown in the operation's OMG IDL specification.

```
CALL "CORBA-SERVERREQUEST-OPERATION" USING
            A-CORBA-SERVERREQUEST
            A-CORBA-ENVIRONMENT
      A-CORBA-IDENTIFIER
```

1.25.9.3 ctx

This function may be used to determine any context values passed as part of the operation. Context will only be available to the extent defined in the operation's OMG IDL definition; for example attribute operations have none.

```
CALL "CORBA-SERVERREQUEST-CTX" USING
            A-CORBA-SERVERREQUEST
            A-CORBA-ENVIRONMENT
      A-CORBA-CONTEXT
```

1.25.9.4 arguments

This function is used to retrieve parameters from the ServerRequest, and to find the addresses used to pass pointers to result values to the ORB. It must always be called by each Dynamic Implementation Routine (DIR), even when there are no parameters.

The caller passes ownership of the parameters NVList to the ORB. Before this routine is called, that NVList should be initialized with the TypeCodes and direction flags for each of the parameters to the operation being implemented: **in**, **out**, and **inout** parameters inclusive. When the call returns, the parameters NVList is still usable by the DIR, and all in and inout parameters will have been unmarshaled. Pointers to those parameter values will at that point also be accessible through the parameters NVList.

The implementation routine will then process the call, producing any result values. If the DIR does not need to report an exception, it will replace pointers to **inout** values in parameters with the values to be returned, and assign parameters to **out** values in that NVList appropriately as well. When the DIR returns, all the parameter memory is freed as appropriate, and the NVList itself is freed by the ORB.

```
CALL "CORBA-SERVERREQUEST-ARGUMENTS" USING
            A-CORBA-SERVERREQUEST
            A-CORBA-NVLIST
      A-CORBA-ENVIRONMENT
```

1.25.9.5 set-result

This function is used to report any result value for an operation. If the operation has no result, it must either be called with a tk-void TypeCode stored in value, or not be called at all.

```
CALL "CORBA-SERVERREQUEST-SET-RESULT" USING
            A-CORBA-SERVERREQUEST
            A-CORBA-ANY
      A-CORBA-ENVIRONMENT
```

1.25.9.6 set-exception

This function is used to report exceptions, both user and system, to the client who made the original invocation.

```
CALL "CORBA-SERVERREQUEST-SET-EXCEPTION" USING
            A-CORBA-SERVERREQUEST
            A-CORBA-EXCEPTION-TYPE
            A-CORBA-ANY
      A-CORBA-ENVIRONMENT
```

The parameters are as follows:

- The exception-type indicates whether it is a USER or a SYSTEM exception.
- the CORBA-any is the value of the exception (including the exception TypeCode).

1.25.10 Mapping of Dynamic Implementation Routine to COBOL

A COBOL Dynamic Implementation Routine will be as follows:

```
PROCEDURE DIVISION USING
      A-PORTABLESERVER-SERVANT
      A-CORBA-SERVERREQUEST
```

Such a function will be invoked by the Portable Object Adapter when an invocation is received on an object reference whose implementation has registered a dynamic skeleton:

- Servant is the COBOL implementation object incarnating the CORBA object to which the invocation is directed.
- Request is the ServerRequest used to access explicit parameters and report results (and exceptions).

Unlike other COBOL object implementations, the DIR does not receive a CORBA-Environment parameter, and so the CORBA-exception-set API is not used. Instead, CORBA-ServerRequest-set-exception is used; this provides the TypeCode for the exception to the ORB, so it does not need to consult the Interface Repository (or rely on compiled stubs) to marshal the exception value.

To register a Dynamic Implementation Routine with a POA, the proper EPV structure and servant must first be created. DSI servants are expected to supply EPVs for both PortableServer-ServantBase and for PortableServer-DynamicImpl, which is conceptually derived from PortableServer-ServantBase, as shown below.

```
01 PORTABLESERVER-DYNAMICIMPL-EPV IS TYPEDEF.
   03 PRIVATE      USAGE POINTER.
   03 INVOKE       TYPE PORTABLESERVER-DYNAMICIMPLROUTINE.
   03 PRIMARY-INTERFACE USAGE PROCEDURE-POINTER.

*  (PRIMARY-INTERFACE SIGNATURE IS AS FOLLOWS ...)
   CALL "PRIMARY-INTERFACE" USING
                     A-PORTABLESERVER-SERVANT
                     A-PORTABLESERVER-OBJECTID
                     A-PORTABLESERVER-POA
                     A-CORBA-ENVIRONMENT
               A-CORBA-REPOSITORYID-RTN

01 PORTABLESERVER-DYNAMICIMPL-VEPV  IS TYPEDEF.
   03 BASE_EPV                         USAGE POINTER
   03 PORTABLESERVER-DYNAMICIMPL-EPV   USAGE POINTER.

01 PORTABLESERVER-DYNAMICIMPL       IS TYPEDEF.
   03 PRIVATE                          USAGE POINTER.
   03 VEPV                             USAGE POINTER.
```

As for other servants, initialization and finalization functions for PortableServer-DynamicImpl are also provided, and must be invoked as described in Section 1.25.6, “Servant Structure Initialization,” on page 1-67.

To properly initialize the EPVs, the application must provide implementations of the invoke and the primary-interface functions required by the PortableServer-DynamicImpl EPV. The invoke method, which is the DIR, receives requests issued to any CORBA object it represents and performs the processing necessary to execute the request.

The primary-interface method receives an ObjectId value and a POA as input parameters and returns a valid Interface Repository Id representing the most-derived interface for that oid.

It is expected that these methods will be only invoked by the POA, in the context of serving a CORBA request. Invoking these methods in other circumstances may lead to unpredictable results.

An example of a DSI-based servant is shown below:

```
IDENTIFICATION DIVISION.
   PROGRAM ID. MY-INVOKE.
   ...
PROCEDURE DIVISION USING
               A-PORTABLESERVER-SERVANT
               A-CORBA-SERVERREQUEST
```

```
    ...
END-PROGRAM.

IDENTIFICATION DIVISION.
   PROGRAM ID. MY-PRIM-INTF.
    ...
PROCEDURE DIVISION USING
               A-PORTABLESERVER-SERVANT
               A-PORTABLESERVER-OBJECTID
               A-PORTABLESERVER-POA
               A-CORBA-ENVIRONMENT
               A-CORBA-REPOSITORYID-RTN
    ...
END-PROGRAM.


/* APPLICATION-SPECIFIC DSI SERVANT TYPE */
01 MYDSISERVANT      IS TYPEDEF.
   03 BASETYPE POA-DYNAMICIMPL.
   ....
   <OTHER APPLICATION SPECIFIC DATA ITEMS>
   ....

01 BASE-EPV          TYPE PORTABLESERVER-SERVANTBASE-EPV.
01 DYNAMICIMPL-EPV   TYPE PORTABLESERVER-DYNAMICIMPL-EPV.
01 DYNAMICIMPL-VEPV  TYPE PORTABLESERVER-DYNAMICIMPL-VEPV.
01 MY-SERVANT        TYPE MYDSISERVANT.
    ...
*  INITIALIZE BASE-EPV
      SET PRIVATE IN BASE-EPV       TO NULL.
      SET FINALIZE IN BASE-EPV      TO NULL.
      SET DEFAULT-POA IN BASE-EPV   TO NULL.
    ...
*  INITIALIZE DYNAMICIMPL-EPV
      SET PRIVATE IN DYNAMICIMPL-EPV   TO NULL.
      SET INVOKE IN DYNAMICIMPL-EPV
                     TO ENTRY "MY-INVOKE".
      SET PRIMARY-INTERFACE IN DYNAMICIMPL-EPV
                     TO ENTRY "MY-PRIM-INTF".
    ...
*  INITIALIZE DYNAMICIMPL-VEPV
      SET BASE-EPV IN DYNAMICIMPL-VEPV
                     TO ADDRESS OF BASE-EPV.
      SET PORTABLESERVER-DYNAMICIMPL-EPV IN DYNAMICIMPL-VEPV
                     TO ADDRESS OF DYNAMICIMPL-EPV.
    ...
*  INITIALIZE MY-SERVANT
      SET PRIVATE IN BASE IN MY-SERVANT   TO NULL.
      SET VEPV    IN BASE IN MY-SERVANT.
                  TO ADDRESS OF DYNAMICIMPL-VEPV.
   ....
```

Registration of the my-servant data structure via the PortableServer-POA-set-servant function on a suitably initialized POA makes the my-invoke DIR function available to handle DSI requests.

1.26 Extensions to COBOL 85

1.26.1 Overview

The following list of extensions to COBOL 85 are used within both the Dynamic COBOL Mapping, and also the Type Specific COBOL Mapping:

- Untyped pointers and pointer manipulation
- Floating point

The following list of extensions to COBOL 85 are only used within the Potable COBOL Mapping:

- Constants
- Typedefs

1.26.2 Untyped Pointers and Pointer Manipulation

1.26.2.1 Untyped Pointers

COBOL 85 does not define an untyped pointer data type. However, the following syntax has been defined within the next major revision of COBOL 85 and has already been implemented in many current COBOL compilers.

```
[ USAGE IS ]       POINTER
```

- [2] No PICTURE clause allowed

1.26.2.2 Pointer Manipulation

COBOL 85 does not define any syntax for the manipulation of untyped pointers. However, the following syntax has been defined within the next major revision of COBOL 85 and has already been implemented in many current COBOL compilers.

```
                                     {ADDRESS OF IDENTIFIER}
SET {ADDRESS OF IDENTIFIER} TO       {IDENTIFIER    }
    {IDENTIFIER           }          {NULL     }
                                     {NULLS    }


                                     {IDENTIFIER   }
SET {IDENTIFIER{UP } }         BY    {INTEGER   }
               {DOWN}                {LENGTH OF IDENTIFIER}
```

1.26.3 Floating point

Currently COBOL 85 does not support floating point data types. There is an implicit use of floating point within this mapping. The OMG IDL floating-point types are specified as follows within the *CORBA* specification:

- float represents single precision floating point numbers
- double represents double-precision floating point numbers
- long double represents long-double-precision floating point numbers

The above IDL types should be mapped to the native floating point type. The ORB will then be responsible for converting the native floating point types to the Common Data Representation (CDR) transfer syntax specified for the OMG IDL floating-point types.

1.26.4 Constants

Currently COBOL 85 does not define any syntax for COBOL constants. The next major revision of COBOL 85 defines the syntax below for this functionality.

To ensure that a complete mapping of CORBA IDL can be accomplished within a COBOL application, it will be necessary to map CORBA IDL constants to some form of COBOL constant such as this.

```
>>CONSTANT CONSTANT-NAME   IS    LITERAL
                                 INTEGER
```

1.26.5 Typedefs

Currently COBOL 85 does not define any syntax for COBOL typedefs. The next major revision of COBOL 85 defines the syntax below for this functionality.

A typedef is defined using the IS TYPEDEF clause on a standard data entry. It identifies it as a typedef and will have no storage associated with it.

It is later used in conjunction with the TYPE clause to identify a user defined data type. The following is an example of this syntax.

```
*   (DEFINES A TYPEDEF)
01 MY-MESSAGE-AREA-TYPE IS TYPEDEF.
   02 WS-LENGTH              USAGE PIC 9(4) COMP.
   02 WS-TEXT                USAGE PIC X(40).

   .....
*   (USING TYPES IN STORAGE DEFINITIONS)
01 WS-MESSAGE1               TYPE MY-MSG-AREA-TYPE.
01 WS-MESSAGE2               TYPE MY-MSG-AREA-TYPE.

   .....
*   (MANIPULATE DATA AS REQUIRED)
```

```
PROCEDURE DIVISION.
   .....
   MOVE 12        TO WS-LENGTH IN WS-MESSAGE1.
   MOVE MSG1      TO WS-TEXT   IN WS-MESSAGE1.
   .....
```

Index

Index