Mapping of OMG IDL to Cobol 22

Contents

This chapter contains the following sections.

22.1 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.

Support

The mapping has been designed to support as many COBOL compilers and ORB implementations as possible. Additionally, it has been designed so that an actual implementation may be based upon the current ANSI COBOL 85 language standard for the COBOL programming language with some additional commonly-used extensions from the next ANSI COBOL language standard.

Currently, the next ANSI COBOL language standard is at a draft stage and will soon be ratified. For a description of the syntax taken from the next draft for use with standard ANSI COBOL 85, refer to "Extensions to COBOL 85" on page 22-49.

22.2 Mapping of IDL to COBOL

22.2.1 Mapping of IDL Identifiers to COBOL

Mapping IDL Identifiers to a COBOL Name

A COBOL name can only be up to 30 characters in length and may consist of a combination of letters, digits, and hyphens. The hyphen cannot 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
```

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.

22.3 Scoped Names

The COBOL programmer must always use the global names 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 global scope will be ignored.

Consider the following example:

```
// IDL

interface Example {

        struct {
                long rtn_code;
                ...
        } return_type;

        return_type my_operation();
        ...
};
```

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

```
PROCEDURE DIVISION.
    ...
    call "Example-my-operation" using
            a-Example-object
            a-CORBA-environment
         a-return-type
    if rtn-code in a-return-type NOT = 0
        ...
    end-if
    ...
```

Care should be taken to avoid ambiguity within COBOL derived from IDL. Consider the following example:

```
typedef long foo_bar;
interface foo {
        typedef short bar;          /* Valid IDL, but ambiguous in COBOL */
};
```

Is **foo-bar** a short or a long in the above example?

Note – It is strongly recommended that you take great care to avoid the use of indiscriminate underscores and hyphens.

22.4 Memory Management

The standard ORB-supplied functions CORBA-alloc and CORBA-free may be used to allocate and free storage for data types. For further details on these functions refer to "Memory Management" on page 22-23.

22.5 Mapping for Interfaces

22.5.1 Object References

The use of an interface type in IDL denotes an object reference. Each IDL interface shall be mapped to the well-known opaque type **CORBA-Object**.

The following example illustrates the COBOL mapping for an interface:

```
interface interface1 {
            ...
};
```

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

```
01 interface1     is typedef           type CORBA-Object.
```

22.5.2 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     is typedef        type CORBA-Object.
    ...
```

The following is a sample of COBOL code that may be used to call **op2**:

```
WORKING-STORAGE SECTION.
...
01 interface1-obj       type interface1.
01 interface2-obj       type interface2.
01 ev                   type CORBA-Environment.
...

PROCEDURE DIVISION.
    ...
    call     “interface2-op2” using
                  interface2-obj
                  ev
             interface1-obj
    ...
```

22.5.3 Inheritance and Interface Names

IDL allows the specification of interfaces that inherit operations from other interfaces. Consider the following example:

```
interface interface3 : interface1 {
      void op3(in long parm3a, out long parm3b);
};
```

A call to either **interface1-op1** or **interface3-op1** on the above **interface3** object will cause the same actual method to be invoked. This is illustrated within the following examples.

CORBA clients, written in COBOL, can make calls to the **op1** operation that was inherited from **interface1** on an **interface3** object as if it had been directly declared within the **interface3** interface:

```
call "interface3-op1" using
         interface3-obj
         aParm1a
         aParm1b
         ev
```

CORBA COBOL clients may also make **interface1-op1** calls on the **interface3** object.

```
call "interface1-op1" using
         interface3-obj
         aParm1a
         aParm1b
         ev
```

22.6 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, 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
         aCORBA-environment
      balance-float
```

```
call “foo--set-balance” using
         a-foo-object
         balance-float
         aCORBA-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 exceptions since the syntax of attribute declarations does not permit them.

22.7 Mapping for Constants

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

Refer to “Extensions to COBOL 85” on page 22-49 for details of the Constant syntax within the next major revision of COBOL.

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-StringIS “This is a string value”.
>>CONSTANT My-Const-NumberIS 100.
```

22.8 Mapping for Basic Data Types

The basic data types have the mappings shown in the following table. Implementations are responsible for providing either COBOL typedefs or COBOL COPY files (whichever is appropriate for their COBOL environment):

- COBOL typedefs for CORBA-short, CORBA-unsigned-short, etc. are consistent with OMG IDL requirements for the corresponding data types. (Note: Support for COBOL Typedefs is an optional extension to ANSI 85 for this mapping).

- COBOL COPY files within a COBOL library named CORBA. The COPY files will contain types that are consistent with OMG IDL requirements for the corresponding data types. (For further details, refer to "Using COBOL COPY files instead of Typedefs" on page 22-51).

Table 22-1 COBOL COPY files within a COBOL library named CORBA

OGM IDL	COBOL Typedef	COBOL COPY file in a CORBA library
short	CORBA-short	short
long	CORBA-long	long
long long	CORBA-long-long	llong
unsigned short	CORBA-unsigned-short	ushort
unsigned long	CORBA-unsigned-long	ulong
unsigned long long	CORBA-unsigned-long-long	ullong
float	CORBA-float	float
double	CORBA-double	double
long double	CORBA-long-double	ldouble
char	CORBA-char	char
wchar	CORBA-wchar	wchar
boolean	CORBA-boolean	boolean
octet	CORBA-octet	octet
enum	CORBA-enum	enum
any	CORBA-any	any

22.8.1 *Boolean*

The COBOL mapping of **boolean** is an integer that may have the values CORBA-true and CORBA-false defined; other values produce undefined behavior. CORBA-boolean is provided for symmetry with the other basic data type mappings.

The following constants will be provided for setting and testing **boolean** types:

```
>>CONSTANT          CORBA-true          is 1.
>>CONSTANT          CORBA-false         is 0.
```

22.8.2 *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         is typedef  type CORBA-enum.
    88 Example-cold                 value 0.
    88 Example-warm                 value 1.
    88 Example-hot                  value 2.
```

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

```
WORKING-STORAGE SECTION.
    ...
01 Example-temp-value   type Example-temp.
    ...
PROCEDURE DIVISION.
    ...
    evaluate TRUE
        when Example-cold of Example-temp-value
            ...
        when Example-warm of Example-temp-value
            ...
        when Example-hot of Example-temp-value
            ...
    end-evaluate
    ...
```

22.8.3 any

The IDL **any** type permits the specification of values that can express any IDL type. The any IDL type will generate the following COBOL group item:

```
01 CORBA-any           is typedef.
    03 any-type            type CORBA-TypeCode.
    03 any-value           usage pointer.
```

For details of TypeCodes, refer to *The Common Object Request Broker: Architecture and Specification*. The IDL-value element of the group item is a pointer to the actual value of the datum.

22.9 Mapping for Fixed Types

For COBOL, the IDL **fixed** type is mapped to the native fixed-point decimal type. The IDL syntax **fixed<digits,scale>** will generate a COBOL typedef that maps directly to the native fixed-point decimal type.

Consider the following example:

```
typedef fixed<9,2> money;
```

The above example describes a fixed point decimal type that contains 9 digits and has a scale of 2 digits (9,999,999.99). It will result in the following COBOL declarations:

```
01 money      is typedef [COBOL fixed point type]
```

22.10 Mapping for Struct Types

IDL structures map directly onto COBOL group items. The following is an example of an IDL declaration of a structure:

```
struct example {
      long                    member1, member2;
      boolean                 member3;
};
```

Would map to the following COBOL:

```
01 <scope>-example      is typedef.
     03 member1         type CORBA-long.
     03 member2         type CORBA-long.
     03 member3         type CORBA-boolean.
```

22.11 Mapping for Union Types

IDL discriminated unions are mapped onto COBOL group items with the **REDEFINES** clause. The following is an example of an IDL declaration of a discriminated union:

```
union example switch(long) {
            case 1:             char first_case;
            case 2:             long second_case;
            default:            double other_case;
};
```

Would map to the following COBOL:

```
01 <scope>-example                is typedef.
     03 d                         type CORBA-long.
     03 u.
```

```
         05 default-case         type CORBA-double.
    03 filler       redefines u.
         05 second-case          type CORBA-long.
    03 filler       redefines u.
         05 first-case           type CORBA-char.
```

The discriminator in the group item is always referred to as **d;** the union items are contained within the group item that is always referred to as **u.**

Reference to union elements is done using standard COBOL. Within the following example, the COBOL “evaluate” statement is used to test the discriminator:

```
evaluate d in <scope>-example
     when 1
       display “Char value = “ first-case in <scope>-example
     when 2
       display “Long value = “ second-case in <scope>-example
     when other
       display “Double value = “ other-case in <scope>-
       example
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 such that the largest is issued first within the generated COBOL structure and the smallest is last (as illustrated within the above example).

22.12 Mapping for Sequence Types

The IDL data type **sequence** permits passing of bounded and unbounded arrays between objects.

Bounded **sequences** are mapped to a typedef that contains an occurs clause up to the specified limit.

For unbounded **sequences**, a pointer to the unbounded array of sequence elements is generated along with a typedef for one sequence element. To access unbounded sequences, two accessor functions are provided (CORBA-sequence-element-get and CORBA-sequence-element-set).

22.12.1 Bounded Sequence

Consider the following bounded IDL **sequence**:

typedef sequence<longfloat,10> vec10;

In COBOL, this is mapped to:

```
01 <scope>-vec10                   is typedef.
     03 seq-maximum                type CORBA-long.
     03 seq-length                 type CORBA-long.
     03 seq-buffer                 usage POINTER.
     03 seq-value    occurs 10     type CORBA-float.
```

For bounded **sequences,** the **seq-buffer** pointer should be set to the address of the **seq-value** item.

22.12.2 Unbounded Sequence

Consider the following unbounded IDL **sequence**:

typedef sequence<long> vec;

In COBOL, this is mapped to the following two typedefs:

```
01 <scope>-vec-t     is typedef        type CORBA-long.

01 <scope>-vec       is typedef.
     03 seq-maximum                    type CORBA-long.
     03 seq-length                     type CORBA-long.
     03 seq-buffer   usage POINTER.    [to <scope>-vec-t]
```

In this case the sequence is unbounded; therefore, a **vec-t** typedef is used to specify one specific instance of the sequence. The **seq-buffer** item should be set to the address of a variable length array of the sequence type.

To access the elements within an unbounded sequence, application developers may either:

- Set up a table of elements of the sequence type within the linkage section using the IDL generated sequence element typedef. Set the table address to the value in **seq-buffer** and use normal table processing logic to step through the elements.
- Use the ORB supplied **sequence** element accessor functions.

22.12.3 Sequence Element Accessor Functions

The following ORB supplied routines may be used to get or set specific elements within a sequence:

```
call “CORBA-sequence-element-get” using
         a-CORBA-sequence
         a-CORBA-unsigned-long
     a-element-type

call “CORBA-sequence-element-set” using
         a-CORBA-sequence
         a-CORBA-long
         a-element-type
```

For further details of the above accessor functions, refer to “Accessor Functions” on page 22-47.

The following is an example of some code that steps through sequence elements using the above “CORBA-sequence-element-get” routine:

```
WORKING-STORAGE SECTION.

01 a-Sequence             type <scope>-vec.
01 ws-vec-element         type <scope>-vec-t.
01 ws-num                 type CORBA-long.
     ...
PROCEDURE DIVISION.
     ...
     PERFORM      VARYING ws-num FROM 1 BY 1
                  UNTIL ws-num > seq-length IN a-Sequence
         call “CORBA-sequence-element-get” using
                   a-Sequence
                   ws-num
                ws-vec-element
         PERFORM process-current-element
     END-PERFORM
     ...
```

22.12.4 Nested Sequences

The type specified within a sequence may be another sequence. In this instance, the generated COBOL declarations are also nested. For example:

typedef sequence<sequence<long> > seq_type;

will be mapped to the following COBOL:

```
01 <scope>-seq-type-t-t is typedef type CORBA-long.

01 <scope>-seq-type-t   is typedef.
     03 seq-maximum type CORBA-long.
     03 seq-length  type CORBA-long.
     03 seq-buffer  usage POINTER.[to <scope>-seq-type-t-t]

01 <scope>-seq-type     is typedef.
     03 seq-maximum type CORBA-long.
     03 seq-length  type CORBA-long.
     03 seq-buffer  usage POINTER. [to <scope>-seq-type-t]
```

22.12.5 Sequence parameter passing considerations

Passing a Sequence as an in parameter

When passing a Sequence as an **in** parameter, the COBOL programmer must:

- set the **buffer** member to point to an array of the specified data type item to point at the allocated storage (or NULL if it is a bounded sequence), and
- set the **length** member to the actual number of elements to transmit.

Passing a Sequence as an out parameter or return

The programmer should pass a pointer (there is no need to initialize it). Once the call has been made, the ORB will have allocated storage for the sequence returned by the object. Upon successful return from the call:

- The **maximum** item will contain the size of the allocated array.
- The **buffer** item will point at the allocated storage (or NULL if it is a bounded sequence).
- The length item will contain the actual number of values that were returned in the **sequence**.

The client is responsible for freeing the allocated sequence by making a call to "CORBA-free" when the returned sequence is no longer required.

Passing a Sequence as an inout parameter

The programmer should pass a pointer to a **sequence** that has been allocated using the CORBA-alloc routine.

Before passing a sequence as an **inout** parameter, the programmer must:

- set the **buffer** item to point to an array buffer (or NULL if it is a bounded sequence), and
- set the length item to the actual number of elements that are to be transmitted.

The CORBA-alloc routine must be used. This allows the callee to deallocate the original **sequence** using a call to "CORBA-free." If more data must be returned, then the original sequence can hold and assign new storage.

Upon successful return from the invocation, the **length** member will contain the returning number of values within the sequence.

For bounded sequences, it is an error to set the **length** or **maximum** item to a value larger than the specified bound.

22.13 Mapping for Strings

22.13.1 How string is mapped to COBOL

Bounded strings

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:

```
typedef string<10> string_1;
```

In COBOL, this is mapped directly to:

```
01 string-1          is typedef   pic x(10).
```

Unbounded strings

An unbounded IDL string cannot be mapped directly to a COBOL PIC X of a specific size, as bounded strings are. Instead, it is mapped to a pointer that is accessed via a set of accessor functions (CORBA-string-get and CORBA-string-set).

Consider the following IDL declarations:

```
typedef string string_2;
```

In COBOL, this is converted to:

```
01 string-2          is typedef   usage POINTER.
```

The following ORB supplied accessor routines may be used to get or set the actual string value:

```
call “CORBA-string-get” using
             a-CORBA-unbounded-string
             a-CORBA-unsigned-long
      a-COBOL-text

call “CORBA-string-set” using
             a-CORBA-unbounded-string
             a-CORBA-unsigned-long
             a-COBOL-text
```

The **CORBA-string-set** routine will be responsible for allocating the storage required and will set the pointer to point to a null terminated string.

The **CORBA-string-get** routine does not release the storage within the pointer; therefore, it may be used more than once to access the same string.

The following is an example of string manipulation using the above routines.

```
WORKING-STORAGE SECTION.
01 my-COBOL-text        pic x(16) value “some random text”.
01 my CORBA-string      type string-2.
     ...

PROCEDURE DIVISION
     ...
     call “CORBA-string-set” using
                    my-CORBA-string
                    LENGTH OF my-COBOL-text
                    my-COBOL-text
     ...
     call “CORBA-string-get” using
                    my-CORBA-string
                    LENGTH OF my-COBOL-text
            my-COBOL-text
     ...
```

For further details of the string accessor routines, refer to “Accessor Functions” on page 22-47.

22.13.2 How wstring is mapped to COBOL

Bounded wstrings

Bounded IDL wstrings are mapped directly to an array of wchar’s of the specified IDL length. The ORB will be totally responsible for handling the null byte, as required. Inbound wstrings will have the null terminator automatically stripped off by the ORB and outbound wstrings will automatically have a null terminator appended by the ORB.

Consider the following IDL declarations:

```
typedef wstring<10> wstring_1;
```

In COBOL, this is mapped to:

```
01 wstring-1-t         is typedef.
     03 filler         type CORBA-wchar occurs 10.
```

Unbounded wstrings

An unbounded IDL wstring cannot be mapped directly to a specific sized area as bounded wstrings are. Instead, it is mapped to a pointer that is accessed via a set of accessor functions (CORBA-wstring-get and CORBA-wstring-set).

Consider the following OMG IDL declarations:

typedef wstring wstring_2

In COBOL, this is converted to:

```
01 wstring-2         is typedef      usage POINTER.
```

The following ORB supplied accessor routines may be used to handle variable length null terminated wstrings:

```
call “CORBA-wstring-get” using
              a-CORBA-wstring
              a-CORBA-unsigned-long
          a-COBOL-wtext

call “CORBA-wstring-set” using
              a-CORBA-wstring
              a-CORBA-unsigned-long
              a-COBOL-wtext
```

The **CORBA-wstring-set** routine will be responsible for allocating the storage required and will return a pointer to a null terminated wstring within the pointer.

The **CORBA-wstring-get** routine does not release the storage within the pointer; therefore, it may be used more than once to access the same wstring.

The following is an example of wstring manipulation using the above routines:

```
WORKING-STORAGE SECTION.
01 my-COBOL-wtext.
    03 filler           type CORBA-wchar occurs 10.
01 my-CORBA-wstring     type wstring-2.
    ...
PROCEDURE DIVISION
    ...
    call “CORBA-wstring-set” using
                  my-CORBA-wstring
                  length of my-COBOL-wtext
                  my-COBOL-wtext
    ...
    call “CORBA-wstring-get” using
                  my-CORBA-wstring
                  length of my-COBOL-wtext
           my-COBOL-wtext
    ...
```

For further details of the string accessor routines, refer to “Accessor Functions” on page 22-47.

22.13.3 string / wstring argument passing considerations

Passing a string or wstring as an in parameter

If the **string /wstring** is bounded, then the COBOL text (or array of double bytes) may be passed directly as an **in** parameter.

If the **string /wstring** is unbounded, a pointer to the null terminated **string/wstring** that was established with the **CORBA-string-set** (or **CORBA-wstring-set**) accessor function is passed.

The accessor function is responsible for the allocation of the storage that the pointer points to. The ORB will be responsible for releasing that storage once it has completed processing the **in** parameter.

The caller is not allowed to pass a null pointer as the string/wstring argument.

Passing a string or wstring as an out parameter or return

If the **string /wstring** is bounded, then the COBOL text (or array of double bytes) is passed back into a COBOL text area supplied by the caller. If necessary, the ORB will be responsible for padding the storage with spaces.

If the **string /wstring** is unbounded, then the pointer to the null terminated **string/wstring** is passed to the caller. The caller uses the appropriate accessor function to obtain the COBOL text value (**CORBA-string-get** or **CORBA-wstring-get**). The caller is responsible for freeing the allocated storage pointed to by the returned pointer using **CORBA-free.**

Passing a string or wstring as an inout parameter

If the **string /wstring** is bounded, then the COBOL text (or array of double bytes) is passed directly as an **in** parameter. The ORB will be responsible for handling the null termination on the user's behalf. Upon return, the COBOL text (or array of double bytes) is passed back to the same area of storage.

The ORB is prohibited from deallocating and reallocating storage for bounded **string/wstring** (the storage is supplied by and belongs to the caller).

If the **string /wstring** is unbounded, the caller must pass a pointer to a null terminated **string/wstring**. The storage is allocated and the value is established within it by using the appropriate accessor function (**CORBA-string-set** or **CORBA-wstring-set).**

The ORB may deallocate and reallocate the buffer if the current buffer size is not large enough to hold the returning string. Upon return, the pointer to the null terminated **string/wstring** is passed to the caller. To obtain the COBOL text value, the caller uses the appropriate accessor function (**CORBA-string-get** or **CORBA-wstring-get**). The caller is then responsible for freeing the allocated storage pointed to by the returned pointer using **CORBA-free.**

22.14 Mapping for Arrays

IDL arrays map to the COBOL OCCURS clause. For example, given the following IDL definition:

```
typedef short ShortArray[2][3][4][5];
```

The COBOL mapping will generate the following:

```
01 <scope>-ShortArray    is typedef.
03 filler                        occurs 2.
    05 filler                    occurs 3.
       07 filler                 occurs 4.
         09 filler               occurs 5.
            11 ShortArray-v         type CORBA-short.
```

22.15 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 identifier for exception>".
```

The identifier for the exception uniquely identifies this exception type. For example, it could be the exception's Interface Repository identifier.

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 the single member are implementation-specific.

22.16 Argument Conventions

22.16.1 Implicit Arguments to Operations

From the COBOL programmer's point of view, 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
```

22.16.2 Argument passing Considerations

All parameters are passed BY REFERENCE.

in parameters

All types are passed directly.

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 have 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.

Example: A 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:

1. Initially, the caller must dynamically allocate storage using the CORBA-alloc function and initialize it directly or use an appropriate accessor function that will dynamically allocate storage (CORBA-xxx-set, where xxx is the type being set up).
2. The pointer to the **inout** parameter is passed.
3. When the call has completed and the user has finished with the returned parameter value, the caller is responsible for deallocating the storage. This is done by making a call to the “CORBA-free” ORB function with the current address in the POINTER.

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:

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

22.16.3 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 22-2 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 22-2 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

22.17 Memory Management

22.17.1 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 22-3 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 22-3 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.”

22.18 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 “Argument Conventions” on page 22-19. One of the implicit parameters is the “**CORBA-Environment”** parameter which is used to pass back exception information to the caller.

22.18.1 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.

22.18.2 Exception Handling Functions

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

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.

```
* COBOL
    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.

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 unitl **CORBA-exception-free()** is called.

```
call “CORBA-exception-id” using
     a-CORBA-environment
     a-pointer
```

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 unitl **CORBA-exception-free()** is called.

```
call “CORBA-exception-value” using
     a-CORBA-environment
     a-pointer
```

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
```

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
```

22.18.3 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 context 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
>>CONSTANT ex-example1  is “<unique example1 identifier>“.
01 MyInterface-example2 is typedef.
>>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
```

22.19 Pseudo Objects

Within the CORBA specification are several interfaces that are pseudo-objects. The differences between a real CORBA object and a pseudo object are as follows:

- There are no servers associated with pseudo objects.
- They are not registered with an ORB.
- References to pseudo-objects are not necessarily valid across computational boundaries.

Pseudo Objects are used by the programmer as if they were ordinary CORBA objects. Because of this, some implementations may choose to implement some of them as real CORBA objects.

22.19.1 Mapping Pseudo Objects to COBOL

Pseudo-objects are mapped from the pseudo-IDL according to the rules specified in the preceding sections of this specification. There are no exceptions to these general mapping rules.

22.19.2 Pseudo-Object mapping example

This section contains a brief example of the mapping of Pseudo-IDL to COBOL.

The following pseudo IDL:

```
module CORBA {

    pseudo interface ORB
    {
            string object_to_string(
                    in Object obj
            );
            ...
    }

    {
}
```

would be mapped to COBOL, as follows:

CORBA-ORB-object-to-string (used to translate an object reference into a string)

```
call "CORBA-ORB-object-to-string" using
                  a-CORBA-ORB
                  a-CORBA-Object
                  a-CORBA-Environment
              a-CORBA-string
```

22.20 Mapping for Object Implementations

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.

22.20.1 Operation-specific Details

This chapter 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.

22.20.2 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:

```
* COBOL
    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.

22.20.3 Mapping for PortableServer::ServantLocator::Cookie

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

```
* COBOL
01 Cookie is typedef    usage POINTER
For the COBOL mapping of the
PortableServer::ServantLocator::preinvoke() and postinvoke()
operations, the Cookie parameter is used as defined
above.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
```

22.20.4 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:

```
* COBOL
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 (note the double underscore). The EPV for **PortableServer-Servant** is defined as follows:

```
* COBOL
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 etherial-ized. 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:

```
* COBOL

*   (vepv is a pointer to the epv)
01 PortableServer-ServantBase-vepv is typedef pointer.
```

```
01 PortableServer-ServantBase is typedef.
    03 privateusage 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 thee next section). The name of the second member, “vepv,” is standardized to allow portable access through it.

22.20.5 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):

```
* COBOL
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:

```
* COBOL
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:

```
* COBOL
01 POA-Counter-vepv is typedef.
     03 base-epv   usage pointer.
     03 Counter-epvusage 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 standarized to allow for portable access to these items.

22.20.6 Servant Structure Initialization

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

```
* COBOL
    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 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 **CORBA-free** 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:

```
* COBOL
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.

22.20.7 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:

```
* COBOL
 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:

* COBOL
IDENTIFICATION DIVISION.
     PROGRAM ID. app-servant-add.
     ...
LINKAGE SECTION.
01 a-AppServant   type 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:

```
* COBOL
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-epvto NULL
     set default-POA in base-epv
                  to ENTRY “my-default-POA”
         ...
*  Initialize counter-epv
     set private in counter-epvto 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-baseto 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-servantto 0
     .
```

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

```
* COBOL
      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()**:

```
* COBOL
                  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.

22.20.8 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:

```
* COBOL
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 5.17.2 Exception Handling Functions) 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 COBOL, it must return a legal function value.

22.21 Mapping of the Dynamic Skeleton Interface to COBOL

Refer to the Dynamic Skeleton Interface chapter for general information about the Dynamic Skeleton Interface (DSI) and its mapping to programming languages.

The following section covers these topics:

- Mapping the ServerRequest Pseudo Object to COBOL
- Mapping the Dynamic Implementation Routine to COBOL

22.21.1 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);
                Any                 set result(any value);
                void                set exception(
                                        exception_type major,
                                                any value
                                     );
        }
}
```

The above ServerRequest pseudo IDL is mapped to COBOL, as follows.

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
     m a-CORBA-Identifier
```

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
     m a-CORBA-Context
```

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 parameter's 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 parameter’s 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 parameter’s NVList.

The implementation routine will then process the call, producing any result values. If the DIR does not have 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-argumentsparams” using
             a-CORBA-ServerRequest
             a-CORBA-NVList
             a-CORBA-Environment
```

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
```

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).

22.21.2 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.

```
* COBOL
01 PortableServer-DynamicImpl-epv      is typedef.
     03 privateusage pointer.
     03 invoketype 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-epvusage 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 “Servant Structure Initialization” in

section 5.19.6. REV???

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:

```
* COBOL
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 base         type 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-epvto NULL.
                  set default-POA in base-epvto NULL.
                    ...
*  Initialize DynamicImpl-epv
                  set private in DynamicImpl-epvto 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-servantto 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.

22.22 ORB Initialization Operations

22.22.1 ORB Initialization

The following PIDL specifies initialization operations for an ORB; this PIDL is part of the CORBA module (not the ORB interface) and is described in Section 7.4, “ORB Initialization,” on page 7-6.

```
// PIDL
module CORBA {
        typedef string ORBid;
        typedef sequence <string> arg_list;
```

```
    ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);
};
```

The mapping of the preceding PIDL operations to COBOL is as follows:

```
* COBOL
01 CORBA-ORBid is typedef type CORBA-string.

01 CORBA-arg-list-t is typedef type CORBA-string.

01 CORBA-arg-list                is typedef.
    03 seq-maximumtype CORBA-long.
    03 seq-length type CORBA-long.
    03 seq-buffer usage POINTER.
                 [to CORBA-arg-list-t]

     call "CORBA-ORB-init" using
             a-CORBA-arg-list
             a-CORBA-ORBid
             a-CORBA-environment
              a-CORBA-ORB
```

If an empty ORBid string is used then arg-list arguments can be used to determine which ORB should be returned. This is achieved by searching the parameter sequence for one tagged ORBid (e.g., -ORBid "ORBid_example"). If an empty ORBid string is used and no ORB is indicated by the **arg-list** parameters, the default ORB is returned.

Regardless of whether an empty or non-empty ORBid string is passed to **ORB_init**, the **argv** arguments are examined to determine if any ORB parameters are given. If a non-empty ORBid string is passed to **ORB_init**, all -ORBid parameters in the **argv** are ignored. All other -ORB<suffix> parameters may be of significance during the ORB initialization process.

22.23 Operations for Obtaining Initial Object References

The following PIDL specifies the operations (in the ORB interface) that allow applications to get pseudo object references for the Interface Repository and Object Services. It is described in detail in Section 7.6, "Obtaining Initial Object References," on page 7-10.

```
// PIDL
module CORBA {
        interface ORB {
         typedef string ObjectId;
         typedef sequence <ObjectId> ObjectIdList;
                        exception InvalidName {};
         ObjectIdList list_initial_services ();
         Object resolve_initial_references (in ObjectId identifier)
                        raises (InvalidName);
        };
};
```

The mapping of the preceding PIDL to COBOL is as follows :

```
* COBOL
01 CORBA-ORB-ObjectId is typedef
                     type CORBA-string.

01 CORBA-ORB-ObjectIdList-t is typedef
                     type CORBA-string.

01 CORBA-ORB-ObjectIdList is typedef.
     03 seq-maximumtype CORBA-long.
     03 seq-length type CORBA-long.
     03 seq-buffer usage POINTER.
                  [to CORBA-ORB-ObjectIdList-t]

01 CORBA-ORB-InvalidName is typedef.
     03 filler[implementation defined]

      call "CORBA-ORB-list-initial-service" using
              a-CORBA-ORB
              a-CORBA-environment
                   a-CORBA-ORB-ObjectIdList-rtn

         call "CORBA-ORB-resolve-initial-refe" using
              a-CORBA-ORB
              a-CORBA-ORB-ObjectId
              a-CORBA-environment
                   a-CORBA-Object-rtn
```

22.24 ORB Supplied Functions for Mapping

22.24.1 Memory Management routines

CORBA-alloc

The ORB supplied **CORBA-alloc** routine may be used to dynamically allocate storage for any of the COBOL data types.

```
call “CORBA-alloc” using
             CORBA-unsigned-long
    m POINTER
```

CORBA-unsigned-long	Specifies the number of bytes of storage to be allocated.
POINTER	Returns address of allocated storage.

CORBA-free

The ORB supplied **CORBA-free** routine may be used to free storage that has previously been dynamically allocated by either the user or the ORB.

```
call “CORBA-free” using
             POINTER
```

POINTER	Address of allocated storage that is to be deallocated.

22.25 Accessor Functions

22.25.1 CORBA-sequence-element-get and CORBA-sequence-element-set

The following ORB supplied routines may be used to get or set specific elements within a **sequence**.

CORBA-sequence-element-get

```
call “CORBA-sequence-element-get” using
         CORBA-sequence
         CORBA-unsigned-long
m     element-type
```

CORBA-sequence	The CORBA-sequence from which a specific element is to be extracted.
CORBA-unsigned-long	An index that identifies the particular element required (1 for the 1st, 2 for the 2nd, etc.).
element-type	An area into which the requested element is to be placed.

CORBA-sequence-element-set

```
call “CORBA-sequence-element-set” using
             CORBA-sequence
```

```
               CORBA-unsigned-long
     melement-type
```

CORBA-sequence	The CORBA-sequence into which a specific element is to be placed.
CORBA-unsigned-long	An index that identifies the particular element (1 for the 1st, 2 for the 2nd, etc.).
element-type	The specific element that is to be inserted into the CORBA-sequence.

22.25.2 CORBA-string-get and CORBA-string-set

The following ORB supplied accessor routines may be used to handle variable length null terminated strings.

CORBA-string-get

```
call “CORBA-string-get” using
               CORBA-unbounded-string
               CORBA-unsigned-long
     mCOBOL-text
```

CORBA-unbounded-string	A pointer to a null terminated string.
CORBA-unsigned-long	The length of the COBOL text area into which the text is to be inserted. The returned value will be truncated (if larger than the return area) or space padded (if smaller than the return area).
COBOL-text	An area into which the requested text is to be placed.

CORBA-string-set

```
call “CORBA-string-set” using
               CORBA-unbounded-string
               CORBA-unsigned-long
               COBOL-text
```

CORBA-unbounded-string	An unintialized pointer into which a null terminated string will be placed by this routine. This routine will use CORBA-alloc to allocate the required storage.
CORBA-unsigned-long	The length of the COBOL text area from which the text is to be extracted. Trailing spaces will be stripped off.
COBOL-text	An area from which the requested text is to be extracted.

22.25.3 CORBA-wstring-get & CORBA-wstring-set

The following ORB supplied accessor routines may be used to handle variable length null terminated wstrings.

```
CORBA-wstring-get
    call “CORBA-wstring-get” using
            CORBA-unbounded-wstring
            CORBA-unsigned-long
        mCOBOL-wchar-values
```

CORBA-unbounded-wstring	A pointer to a null terminated wstring.
CORBA-unsigned-long	The length of the area into which the array of wchars is to be inserted. The returned value will be truncated (if larger than the return area) or padded (if smaller than the return area).
COBOL-wchar-values	An area into which the requested COBOL wchars are to be placed.

CORBA-wstring-set

```
call “CORBA-wstring-set” using
            CORBA-unbounded-string
            CORBA-unsigned-long
            COBOL-wchar-values
```

CORBA-unbounded-wstring	An unintialized pointer into which a null terminated wstring will be placed by this routine. This routine will use CORBA-alloc to allocate the required storage.
CORBA-unsigned-long	The length of the COBOL area from which the wchars are to be extracted.
COBOL-wchar-values	An area from which the requested wchars are to be extracted.

22.26 Extensions to COBOL 85

The following extensions to COBOL 85 are **mandatory** within this submission:

- Untyped pointers and pointer manipulation
- Floating point

The following extensions to COBOL 85 are **optional** within this submission:

- Constants
- Typedefs

22.26.1 Untyped Pointers and Pointer manipulation

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 current COBOL compilers.

```
    [ USAGE IS ]      POINTER
```

No PICTURE clause allowed.

22.26.2 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    }
```

22.26.3 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 CORBA:

- **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.

22.26.4 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.

```
>>CONSTANT constant-name         IS literal
                                    integer
```

22.26.5 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.
.....
```

Using COBOL COPY files instead of Typedefs

Because COBOL typedefs are an optional part of this language mapping, an alternative to the functionality provided by them is part of this COBOL language mapping. While it is recognized that support for COBOL Typedefs is very desirable, it must also be recognized that such support is not yet available from some of the older COBOL compilers deployed on some platforms. It is highly recommended that, if at all possible, COBOL Typedefs should be used because no other alternative offers the same flexibility.

For compilers that do not support COBOL Typedefs, libraries of COBOL COPY files will be used instead. Each library will contain a set of COPY files for each interface, and each individual COPY file will act as a type template for defined IDL data types. When used in conjunction with the COPY REPLACING syntax, the COPY files may be used to create specific instances of types.

How do libraries of COBOL COPY files containing IDL data type templates work?

For basic types, such as **long,** a COPY file called **long** will be supplied as part of a **CORBA** library and its contents would resemble the following:

```
long-type          usage (local long type).
```

The user would use the above long copy file to create instances of the basic long type, as follows:

```
WORKING STORAGE section.
    ...
01 COPY LONG      IN    CORBA
        REPLACING long-type WITH ws-long-1.
01 COPY LONG      IN    CORBA
        REPLACING long-type WITH ws-long-2.
    ...
```

Each specific IDL file will result in a library of COPY files for all the types specified within the interface file.

For example, the following IDL:

```
// IDL
    interface Example {

            struct {
                    long a_long_value;
                    float a_float_value;
            } struct_1;
            ...
            struct {
                    struct_1 a_struct_1_value;
                    long another_long;
            } struct_2;
    };
```

Would result in COPY files called **struct-1** and **struct-2** being created in a library called **Example.**

The following illustrates the contents of the **struct-1** copy file:

```
struct-1-type.
    05 COPY long IN corba
        REPLACING long-type       WITH a-long-value.
    05 COPY float IN corba
         REPLACING float-type       WITH a-float-value.
```

One problem with COPY file templates is that it is not possible to embed a struct template within another struct because of level number resolution problems. Within a user application, it will only be possible to create level 01 instances of structures. This is resolved by generating the actual definitions all the way down to basic types within each generated COPY file. From the above IDL, the following example of **struct_2** illustrates this:

```
struct-2-type
    05 struct-1-value.
         07 COPY long IN corba
              REPLACING long-type    WITH a-long-value.
         07 COPY float IN corba
              REPLACING float-type   WITH a-float-value.
    05 COPY long IN corba
         REPLACING long-type        WITH another-long.
```

22.27 References

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