USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF
MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.
IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND
Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 250 First Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS
The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI® and IIOP® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBA services™, CORBAfacilities™, CORBAmed™, CORBAnet™, Integrate 2002™, Middleware That's Everywhere™, UML™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ and the XMI Logo™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE
The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

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 listed on the main web page http://www.omg.org, under Documents & Specifications, Report a Bug/Issue.
Contents

Preface ......................................................... v

1. Overview ................................................. 1-1
   1.1 Introduction .......................................... 1-1
   1.2 Identifier Names ...................................... 1-2
       1.2.1 Example .......................................... 1-2

2. PL/I Language Mapping ................................. 2-1
   2.1 Mapping for Basic Types ............................ 2-2
       2.1.1 Boolean .......................................... 2-3
       2.1.2 Enum ............................................ 2-3
       2.1.3 Octet and Char ................................. 2-3
   2.2 Mapping for Strings ................................ 2-4
       2.2.1 Bounded Strings ............................... 2-4
       2.2.2 Unbounded Strings ............................ 2-4
   2.3 Mapping for Fixed Types ......................... 2-5
   2.4 Mapping for Struct Types ....................... 2-6
   2.5 Mapping for Union Types ......................... 2-6
   2.6 Mapping for Sequence Types ...................... 2-7
       2.6.1 Bounded ........................................ 2-7
       2.6.2 Unbounded .................................... 2-8
   2.7 Mapping for Arrays ................................. 2-9
   2.8 Mapping for Anys .................................. 2-10
   2.9 Mapping for Exceptions ........................... 2-11
   2.10 Mapping for Interfaces ........................... 2-13
Contents

2.11 Mapping for Typedefs ............................................. 2-13
2.12 Mapping for Valuetypes ......................................... 2-14
  2.12.1 Valuetype Data Members ................................. 2-14
  2.12.2 Valuetype Data Example ............................... 2-15
  2.12.3 Valuetype Operations ................................. 2-16
  2.12.4 Valuetype Example ................................. 2-16
  2.12.5 Generic Valuetype Procedures ....................... 2-19
  2.12.6 Value Boxes ........................................ 2-20
  2.12.7 Value Box Example ................................ 2-20
  2.12.8 Abstract Valuetsypes .............................. 2-22
  2.12.9 Valuetype Inheritance .............................. 2-22
2.13 Portable Object Adapter Mapping ............................ 2-24
  2.13.1 PortableServer Procedures .......................... 2-25
  2.13.2 Mapping for PortableServer::ServantLocator::Cookie ........................................ 2-25
  2.13.3 PortableServer::Servant Mapping ..................... 2-26
  2.13.4 Interface Skeletons .................................. 2-27
  2.13.5 Servant Structure Initialization ..................... 2-28
  2.13.6 Application Servants ................................ 2-30
  2.13.7 Method Signatures .................................... 2-32
2.14 Miscellaneous Other Mappings (Generator Optional) .... 2-32
2.15 Memory Handling by the POD .................................. 2-33
  2.15.1 Unbounded Sequences .................................. 2-33
  2.15.2 Unbounded Sequences and Memory Management .......... 2-33
  2.15.3 Unbounded Strings ..................................... 2-34
  2.15.4 Unbounded strings and memory management ......... 2-35
  2.15.5 In Summary ............................................ 2-35
2.16 POD Function Summary .......................................... 2-35
  2.16.1 Object Invocation ..................................... 2-40
  2.16.2 Server Implementation ................................ 2-42
  2.16.3 ANYFREE ........................................... 2-45
  2.16.4 ANYGET ............................................. 2-46
  2.16.5 ANYSET ............................................. 2-47
  2.16.6 MEMALLOC ........................................... 2-48
  2.16.7 MEMDEBUG ........................................... 2-48
  2.16.8 MEMFREE ............................................ 2-49
  2.16.9 OBJ2STR ............................................ 2-49
  2.16.10 OBJGTID ............................................. 2-50
  2.16.11 OBJNEW ............................................. 2-51
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.16.12</td>
<td>OBJRIR</td>
<td>2-51</td>
</tr>
<tr>
<td>2.16.13</td>
<td>ORBARGS</td>
<td>2-52</td>
</tr>
<tr>
<td>2.16.14</td>
<td>PODERR</td>
<td>2-53</td>
</tr>
<tr>
<td>2.16.15</td>
<td>PODEXEC</td>
<td>2-54</td>
</tr>
<tr>
<td>2.16.16</td>
<td>PODGET</td>
<td>2-55</td>
</tr>
<tr>
<td>2.16.17</td>
<td>PODINFO</td>
<td>2-56</td>
</tr>
<tr>
<td>2.16.18</td>
<td>PODINIT</td>
<td>2-57</td>
</tr>
<tr>
<td>2.16.19</td>
<td>PODPUT</td>
<td>2-57</td>
</tr>
<tr>
<td>2.16.20</td>
<td>PODREG</td>
<td>2-58</td>
</tr>
<tr>
<td>2.16.21</td>
<td>PODREGI</td>
<td>2-59</td>
</tr>
<tr>
<td>2.16.22</td>
<td>PODREQ</td>
<td>2-60</td>
</tr>
<tr>
<td>2.16.23</td>
<td>PODSRVR</td>
<td>2-61</td>
</tr>
<tr>
<td>2.16.24</td>
<td>PODSTAT</td>
<td>2-61</td>
</tr>
<tr>
<td>2.16.25</td>
<td>SEQALOC</td>
<td>2-62</td>
</tr>
<tr>
<td>2.16.26</td>
<td>SEQUPL</td>
<td>2-63</td>
</tr>
<tr>
<td>2.16.27</td>
<td>SEQFREE</td>
<td>2-64</td>
</tr>
<tr>
<td>2.16.28</td>
<td>SEQGET</td>
<td>2-65</td>
</tr>
<tr>
<td>2.16.29</td>
<td>SEQINIT</td>
<td>2-66</td>
</tr>
<tr>
<td>2.16.30</td>
<td>SEQLEN</td>
<td>2-67</td>
</tr>
<tr>
<td>2.16.31</td>
<td>SEQSET, STR2OBJ</td>
<td>2-68</td>
</tr>
<tr>
<td>2.16.32</td>
<td>SEQMAX</td>
<td>2-68</td>
</tr>
<tr>
<td>2.16.33</td>
<td>SEQQET</td>
<td>2-69</td>
</tr>
<tr>
<td>2.16.34</td>
<td>STRCON</td>
<td>2-70</td>
</tr>
<tr>
<td>2.16.35</td>
<td>STRDUPL</td>
<td>2-71</td>
</tr>
<tr>
<td>2.16.36</td>
<td>STRFREE</td>
<td>2-72</td>
</tr>
<tr>
<td>2.16.37</td>
<td>STRGET</td>
<td>2-72</td>
</tr>
<tr>
<td>2.16.38</td>
<td>STRLEN</td>
<td>2-73</td>
</tr>
<tr>
<td>2.16.39</td>
<td>STRSET, STRSETS</td>
<td>2-73</td>
</tr>
<tr>
<td>2.16.40</td>
<td>TYPEGET</td>
<td>2-74</td>
</tr>
<tr>
<td>2.16.41</td>
<td>TYPESET</td>
<td>2-75</td>
</tr>
<tr>
<td>2.16.42</td>
<td>WSTRCON</td>
<td>2-76</td>
</tr>
<tr>
<td>2.16.43</td>
<td>WSTRDUPL</td>
<td>2-76</td>
</tr>
<tr>
<td>2.16.44</td>
<td>WSTRFREE</td>
<td>2-77</td>
</tr>
<tr>
<td>2.16.45</td>
<td>WSTRGET</td>
<td>2-77</td>
</tr>
<tr>
<td>2.16.46</td>
<td>WSTRL EN</td>
<td>2-78</td>
</tr>
<tr>
<td>2.16.47</td>
<td>WSTFRSET, WSTFRSTS</td>
<td>2-78</td>
</tr>
</tbody>
</table>

**Appendix A - Check Errors** ........................................... A-1
Preface

About This Document

Under the terms of the collaboration between OMG and The Open Group, this document is a candidate for adoption by The Open Group, as an Open Group Technical Standard. The collaboration between OMG and The Open Group ensures joint review and cohesive support for emerging object-based specifications.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization’s charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG’s objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based. More information is available at http://www.omg.org/.

The Open Group

The Open Group, a vendor and technology-neutral consortium, is committed to delivering greater business efficiency by bringing together buyers and suppliers of information technology to lower the time, cost, and risks associated with integrating new technology across the enterprise.
The mission of The Open Group is to drive the creation of boundaryless information flow achieved by:

- Working with customers to capture, understand and address current and emerging requirements, establish policies, and share best practices;
- Working with suppliers, consortia and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate specifications and open source technologies;
- Offering a comprehensive set of services to enhance the operational efficiency of consortia; and
- Developing and operating the industry’s premier certification service and encouraging procurement of certified products.

The Open Group has over 15 years experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification. The Open Group portfolio of test suites includes tests for CORBA, the Single UNIX Specification, CDE, Motif, Linux, LDAP, POSIX.1, POSIX.2, POSIX Realtime, Sockets, UNIX, XPG4, XNFS, XTI, and X11. The Open Group test tools are essential for proper development and maintenance of standards-based products, ensuring conformance of products to industry-standard APIs, applications portability, and interoperability. In-depth testing identifies defects at the earliest possible point in the development cycle, saving costs in development and quality assurance.


**About CORBA Language Mapping Specifications**

The CORBA Language Mapping specifications contain language mapping information for the several languages. Each language is described in a separate stand-alone volume.

**Alignment with CORBA**

This language mapping is aligned with CORBA, v2.3.1.

**Associated OMG Documents**

The CORBA documentation is organized as follows:

- *Object Management Architecture Guide* defines the OMG’s technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- **CORBA Services: Common Object Services Specification** contains specifications for OMG’s Object Services.

- **CORBA Common Facilities:** contains services that many applications may share, but which are not as fundamental as the Object Services.

- CORBA domain specifications are comprised of stand-alone documents for each specification; however, they are listed under the domain headings, such as Telecoms, Finance, Med, etc.

OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote.

To obtain OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters  
250 First Avenue  
Needham, MA 02494  
USA  
Tel: +1-781-444-0404  
Fax: +1-781-444-0320  
pubs@omg.org  
http://www.omg.org

**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.

**Typographical Conventions**

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

**Helvetica bold** - OMG Interface Definition Language (OMG IDL) and syntax elements.
Courier bold - Programming language elements.

Helvetica - Exceptions

If applicable, terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Acknowledgments

The following companies submitted parts of this specification:

- International Business Machines Corporation
- IONA Technologies, PLC.
Overview

Contents

This chapter contains the following sections.

<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Introduction”</td>
<td>1-1</td>
</tr>
<tr>
<td>“Identifier Names”</td>
<td>1-2</td>
</tr>
</tbody>
</table>

Alignment

This specification gives a PL/I language mapping for IDL types and functionality. It is aligned with CORBA 2.3.1.

1.1 Introduction

Due to the lack of an accepted PL/I standard, best practice of PL/I implementation and usage on mainframe systems has been chosen as baseline for the design of this PL/I language mapping. The mapping specification has been verified by an implementation using the VisualAge PL/I compiler (V2R2 for OS/390 (CEESG011) / v2.1.6 or higher for Windows NT). At the time of this writing this, appears to be the most widely used PL/I compiler.

If a mapping is to be implemented for an implementation of the PL/I language, which is single-threaded only, or cannot be guaranteed to be thread-safe, the PL/I mapping implementation shall be single-threaded only, and the POA ThreadPolicy shall only accept the value of SINGLE_THREAD_MODEL. Other values shall raise the InvalidPolicy exception.
If the underlying PL/I implementation is known to be thread-safe or multi-threaded, the POA ThreadPolicy shall have the standard default of ORB_CTRL_MODEL. Note that all auxiliary functions must be implemented by thread-safe code in this case.

### 1.2 Identifier Names

All PL/I identifiers may be up to 100 characters in length and must begin with an alphabetic character or an underscore. The following rule is used to convert IDL identifiers to PL/I identifiers and all other names constructed by the mapping (for example, valuetype names):

- It shall be prefixed by the interface name and module name(s) (each separated by an underscore) to ensure its uniqueness.
- The case of the identifiers is preserved, as PL/I is case insensitive.
- If an identifier is greater than 100 characters, it shall be truncated. If this resulting identifier is not unique, then the last five characters shall be replaced with an underscore and a four hex-digit suffix. The suffix shall be obtained via any hash algorithm which shall produce the same results each time for the same identifier name.

Both attributes and operation arguments shall be mapped to a PL/I structure (shown below in the mappings for the various types). Attributes shall be suffixed `'_attr'` and operation arguments `'_args'` at the 1 level (for example, `DCL 1 MY_ARGS,...`). Other suffixes used are shown for the various types below, where appropriate. Note that PL/I is case-insensitive, therefore the suffixes may be presented in any case variation.

#### 1.2.1 Example

```pli
const char myGlobalChar='c';
module m1 {
    interface i1 {
        attribute short aShortVariable;
        void anOperation();
    };
};
```

The identifier’s above shall be mapped as follows:

- `myGlobalChar` → `myGlobalChar_const`
- `aShortVariable` → `m1_i1_aShortVariable_attr`
- `anOperation` → `m1_i1_anOperation_args`
Contents

This chapter contains the following sections.

<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Mapping for Basic Types”</td>
<td>2-2</td>
</tr>
<tr>
<td>“Mapping for Strings”</td>
<td>2-4</td>
</tr>
<tr>
<td>“Mapping for Fixed Types”</td>
<td>2-5</td>
</tr>
<tr>
<td>“Mapping for Struct Types”</td>
<td>2-6</td>
</tr>
<tr>
<td>“Mapping for Union Types”</td>
<td>2-6</td>
</tr>
<tr>
<td>“Mapping for Sequence Types”</td>
<td>2-7</td>
</tr>
<tr>
<td>“Mapping for Arrays”</td>
<td>2-9</td>
</tr>
<tr>
<td>“Mapping for Anys”</td>
<td>2-10</td>
</tr>
<tr>
<td>“Mapping for Exceptions”</td>
<td>2-11</td>
</tr>
<tr>
<td>“Mapping for Typedefs”</td>
<td>2-13</td>
</tr>
<tr>
<td>“Mapping for Valuetypes”</td>
<td>2-14</td>
</tr>
<tr>
<td>“Portable Object Adapter Mapping”</td>
<td>2-24</td>
</tr>
<tr>
<td>“Miscellaneous Other Mappings (Generator Optional)”</td>
<td>2-32</td>
</tr>
<tr>
<td>“Memory Handling by the POD”</td>
<td>2-33</td>
</tr>
<tr>
<td>“POD Function Summary”</td>
<td>2-35</td>
</tr>
</tbody>
</table>
2.1 Mapping for Basic Types

Basic IDL types are mapped as follows. The CORBA type name is given for reference purposes only, the PL/I representation is used directly. For each type which does not have a direct equivalent in PL/I (for example, CORBA-Any), an alias to the PL/I representation is used to clarify it. By doing this, a PL/I structure which contains several types all mapped to the same PL/I representation (for example, unbounded strings and anys) are easily distinguishable.

For retrieving / setting types other than simple types, accessor functions are required. These are described as necessary with each complex type.

<table>
<thead>
<tr>
<th>IDL name</th>
<th>CORBA typedef name</th>
<th>PL/I representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>CORBA-short</td>
<td>FIXED BIN(15)</td>
</tr>
<tr>
<td>long</td>
<td>CORBA-long</td>
<td>FIXED BIN(31)</td>
</tr>
<tr>
<td>unsigned short</td>
<td>CORBA-unsigned-short</td>
<td>UNSIGNED FIXED BIN(16)</td>
</tr>
<tr>
<td>unsigned long</td>
<td>CORBA-unsigned-long</td>
<td>UNSIGNED FIXED BIN(32)</td>
</tr>
<tr>
<td>float</td>
<td>CORBA-float</td>
<td>FLOAT DEC(6)</td>
</tr>
<tr>
<td>double</td>
<td>CORBA-double</td>
<td>FLOAT DEC(16)</td>
</tr>
<tr>
<td>char</td>
<td>CORBA-char</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>boolean</td>
<td>CORBA-boolean</td>
<td>TYPE(BOOLEAN)</td>
</tr>
<tr>
<td>octet</td>
<td>CORBA-octet</td>
<td>TYPE(OCTET)</td>
</tr>
<tr>
<td>enum</td>
<td>CORBA-enum</td>
<td>ORDINAL</td>
</tr>
<tr>
<td>fixed&lt;d,s&gt;</td>
<td>Fixed&lt;d,s&gt;</td>
<td>FIXED DEC(d,s)</td>
</tr>
<tr>
<td>any</td>
<td>CORBA-any</td>
<td>TYPE(ANY)</td>
</tr>
<tr>
<td>long long</td>
<td>CORBA-long-long</td>
<td>FIXED BIN(63)</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>CORBA-unsigned-long-long</td>
<td>UNSIGNED FIXED BIN(64)</td>
</tr>
<tr>
<td>wchar</td>
<td>CORBA-wchar</td>
<td>WIDECHAR(1)</td>
</tr>
</tbody>
</table>

1 Booleans are aliased to a character which have a value of either ‘0’ or ‘1’.
2 Octets are aliased to a character.
3 Anys are aliased to a pointer.
4 Will be supported by VisualAge PL/I in late 2000.
2.1.1 Boolean

A boolean maps to a character data item. Two named constants representing the true and false values shall be provided. The following example illustrates the mapping for booleans.

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

maps to the following PL/I

```plaintext
/* Provided name constants */
declare alias boolean char;
declare corba_false char value('0');
declare corba_true char value('1');

/* Generated variables */
declare 1 example_full_args aligned,
       3 result type(boolean);
```

2.1.2 Enum

An enum is mapped to an `ORDINAL` defined constant. The following example illustrates the mapping of enums.

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

maps to the following PL/I

```plaintext
define ordinal temp(cold, warm, hot);
```

and this may be used as follows:

```plaintext
declare todays_temp ordinal temp;
if todays_temp = cold then
    put skip list('brrrr');
```

2.1.3 Octet and Char

The native encoding of CHAR data values in PL/I depends on the underlying platform (ASCII on Windows NT and Unix, EBCDIC on the mainframe). The mapping implementation shall ensure the correct conversion to and from the common data representation on the wire.

The octet data type shall never undergo such encoding and shall always be transmitted as-is.
2.2 Mapping for Strings

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

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

In PL/I bounded and unbounded strings are represented differently. Unbounded strings are represented by a pointer. Bounded strings are represented by a CHAR(n) data item, where “n” is the bounded length of the string. Note that the maximum length of a bounded string in PL/I is 32,767 characters. Due to how strings are represented in PL/I, accessor functions are used for handling unbounded strings.

The two types of string shall be handled as follows.

2.2.1 Bounded Strings

Retrieving IN and INOUT Values

A bounded string is represented by a CHAR(n) data item.

Sending INOUT, OUT and Return Values

The bounded string is copied out of the buffer. Trailing spaces up to the first null character found in the bounded string are copied (the null itself is not sent). If no nulls are found, the entire string including all trailing spaces is copied.

2.2.2 Unbounded Strings

Retrieving IN and INOUT Values

An unbounded string is represented as a pointer data item, aliased to USTRING for clarity. A pointer is supplied that refers to an area of memory that contains the string data. This is not directly accessible - the STRGET (STRING GET) auxiliary function must be called to copy the data into a CHAR(n) data type (as the length of the unbounded string ‘name’ is not known in advance). For example,

```pli
/* This is the supplied PL/I unbounded string pointer. */
declare name type(ustring); /* ustring = pointer */

/* This is the PL/I representation of the string */
declare supplier_name char (64);

/* This STRGET call copies the characters in NAME to SUPPLIER_NAME */
call strget(name,length,supplier_name);
```

If for some reason the string that is actually passed is too big for this buffer, a MARSHALL::LENGTH_TOO_LONG exception shall be raised and the string shall remain unchanged. If the string is not big enough to fill the buffer, then the balance of the PL/I string is space filled.
Sending INOUT, OUT and Return Values

A valid unbounded string must be supplied by the implementation of an operation. This can be either a pointer that was obtained by an IN/INOUT parameter, or a string constructed using the supplied STRSET (STRING SET) auxiliary function. The following example illustrates this.

/* This is the PL/I representation of the string containing a */
/* value that we want to pass back to the client using PODPUT */
/* via an unbounded pointer string */
declare notes char (160);

/* This is the unbounded pointer string */
declare cust_notes type(ustring);

/* This STRSET call creates a copy of the string in the NOTES */
/* field and assigns the pointer value to */
call strset(cust_notes,length,notes);

Note that trailing space characters are stripped off the string that is constructed with STRSET. This is usually the desired behavior. However if the STRSETS() function (STRING SET with Spaces) is used instead (same argument signature), then exactly the indicated number of characters are copied including the trailing spaces.

2.3 Mapping for Fixed Types

The IDL fixed type maps directly to PL/I packed decimal data with the appropriate number of digits and decimal places if any.

Interface example {
    attribute fixed<5,2> salary;
    attribute fixed<4,4> taxrate;
    attribute fixed<3,-6> millions;
    attribute fixed<3,5> small;
};

maps to the following PL/I (comments added for clarity)

/* Attribute: fixed<5,2> salary */
declare 1 example_salary_args aligned,
        3 result fixed dec(5,2);

/* Attribute: fixed<4,4> taxrate */
declare 1 example_taxrate_args aligned,
        3 result fixed dec(4,4);

/* Attribute: fixed<3,-6> millions */
declare 1 example_millions_args aligned,
        3 result fixed dec(3,-6);
/* Attribute: fixed<3,5> small */
declare 1 example_small_args aligned,
3 result    fixed dec(3,5);

2.4 Mapping for Struct Types

An IDL structure definition maps directly to a PL/I structure. Note that **ALIGNED** is always specified in the top level of the structure.

```
interface example {
  struct mystruct {
    long member1;
    unsigned long member2;
    boolean member3;
  };
  attribute mystruct test;
};
```

maps to the following PL/I

```
declare 1 example_test_args aligned,
3 result,
  5 member1  fixed bin(31),
  5 member2  unsigned fixed bin(32),
  5 member3  type(boolean);
```

2.5 Mapping for Union Types

An IDL union definition such as

```
interface example {
  union un switch(short) {
    case 1: char case_1;
    case 2: double case_2;
    default: long def_case;
  };
  attribute un test;
};
```

maps to the following segment PL/I code

```
declare 1 example_test_args aligned,
3 result,
  5 d fixed bin(15),
  5 uunion,
    7 case_1  char(1),
    7 case_2  float dec(16),
    7 def_case fixed bin(31);
```
The union discriminator in the struct is always referred to as $D$ and the union item storage area is always referred to as $U$. The union items are declared as part of the structure after the `UNION` statement. The storage allocated for the union elements is that of the largest element in the union (`FLOAT DEC(16)` in the above case). Reference to the union elements is done through the `SELECT` statement to test the discriminator, as shown below.

```pli
select(example_test_args.d);
  when(1)
    display('Char value is ' ||
    example_test_args.result.u.case_1);
  when(2)
    display('Long value is ' ||
    example_test_args.result.u.case_2);
  otherwise
    display('Double value is ' ||
    example_test_args.result.u.def_case);
end;
```

Note that the union discriminator may only be of type CHAR, BOOLEAN (which is an aliased CHAR), INTEGER, or ORDINAL.

### 2.6 Mapping for Sequence Types

The PL/I mapping for sequences differs depending on whether the sequence is bounded or unbounded. In both cases however, a supporting pointer is generated which contains the information about the sequence such as the maximum length (accessed via `SEQMAX`) and the length of the sequence (in elements, accessed via `SEQLEN`) and the contents of the sequence (in the case of an unbounded sequence). After a sequence is initialized, the sequence length is equal to zero. Note that the first element of a sequence is referenced as element 1. The `_DAT` suffix contains the actual sequence data.

#### 2.6.1 Bounded

Bounded sequences map to a PL/I array and a supporting data item.

```pli
interface example {
  typedef sequence<long, 10> long10;
  attribute long10 myseq;
};
```

maps to the following PL/I

```pli
define alias seq_ctl pointer;
declare 1 example_myseq_args aligned,
  3 result,
    5 result_seq type(seq_ctl),
    5 result_dat(10) fixed bin(31);
```
2.6.2 Unbounded

Unbounded sequences cannot map to a PL/I array because the size of the sequence is not known. In this case a data item is created to hold one element (suffixed _BUF) of the sequence and a supporting pointer to the sequence’s elements is also created.

```pli
interface example {
    typedef sequence<long, 10> long10;
    attribute long10 myseq;
};
```

maps to the following PL/I

```pli
declare 1 example_myseq_args aligned,
    3 result,
    5 result_seq type(seq_ctl),
    5 result_buf fixed bin(31);
```

Initial storage is assigned to the sequence via `SEQALLOC`. Elements of an unbounded sequence are not directly accessible. Access to specific elements in the sequence is done using the `SEQGET` and `SEQSET` routines, the length of the sequence is found using `SEQLEN` and the maximum length of the sequence is found by a call to the `SEQMAX` function.

2.6.2.1 PODGET - IN and INOUT modes

An unbounded sequence is represented as a pointer data item. A pointer is supplied that refers to an area of memory that contains the sequence. This is not directly accessible - the `SEQGET` auxiliary function must be called to copy a specified element of the sequence into a accessible data area.

The following example, based on the above IDL, walks through all the elements of a sequence.

```pli
declare 1 example_myseq_args aligned,
    3 result,
    5 result_seq type(seq_ctl),
    5 result_buf fixed bin(31);

declare element_num fixed bin(31);
declare result_seq_len fixed bin(31);
.
.
call seqlen(result_seq,result_seq_len);

do element_number = 1 to result_seq_len;
    call seqget(result_seq,element_num,addr(result_buf));
    call process_seq_entry(result_buf);
end;
.
.
```
2.6.2.2  **PODPUT - OUT, INOUT and result only**

A valid unbounded sequence must be supplied by the implementation of an operation. This can be either a pointer that was obtained by an IN/INOUT parameter, or an unbounded sequence constructed using the **SEQALLOC** function.

The **SEQSET** function is used to change the contents of a sequence element. Based on the above example, the following code could be used to store some initial values into all elements of the sequence.

```
declare 1 example_myseq_args ALIGNED,
   3 RESULT,
      5 RESULT_SEQ    TYPE(SEQ_CTL),
      5 RESULT_BUF    FIXED BIN(31);

DECLARE ALLOC_SIZE    FIXED BIN(31);
DECLARE ELEMENT_NUM   FIXED BIN(31);
DECLARE RESULT_SEQ_LEN FIXED BIN(31);

CALL SEQLEN(RESULT_SEQ,RESULT_SEQ_LEN);
ALLOC_SIZE = RESULT_SEQ_LEN * 4;
CALL SEQALLOC(RESULT_SEQ,ALLOC_SIZE,
              CORBA_TYPE_LONG,LENGTH(CORBA_TYPE_LONG));

CALL SEQLEN(RESULT_SEQ,RESULT_SEQ_LEN);

DO ELEMENT_NUMBER = 1 TO RESULT_SEQ_LEN;
   CALL PREPROCESS_SEQUENCE_ENTRY(RESULT_BUF);
   CALL SEQSET(RESULT_SEQ,ELEMENT_NUM,ADDR(RESULT_BUF));
END;
```

2.7  **Mapping for Arrays**

An IDL array definition maps directly to a PL/I array. Each element of the array is directly accessible. It should be noted that PL/I arrays are 1-indexed, not 0-indexed as in C and C++.

```
interface example {
   attribute long long_array[2][5];
};
```

maps to the following PL/I

```
declare 1 example_long_array_args,
   3 result(2,5)    fixed bin(31);

example_long_array_args.result(1,3) = 22;
```
2.8 Mapping for Anys

The IDL `any` type maps to a PL/I structure that provides information about the contents of the any such as the type of the contents. In addition a separate character data item is also generated that is large enough to hold the longest type code string defined in the interface. The contents of the `any` cannot be accessed directly.

```
interface example {
    attribute any temp;
};
```
/maps to the following PL/I

define alias any pointer;

declare 1 example_temp_any_args aligned, 3 result type(any);

/* Assuming that the longest type code is 11 characters long */
declare example_type_codechar(11);

The auxiliary functions `ANYGET` and `ANYSET` are provided to extract data from and insert data into an `any`.

The type of the `any` type can be retrieved using the `TYPEGET` auxiliary function. A data item is generated that can be used to retrieve the type name into. This data item is long enough to hold the largest type name defined in the interface.

The `any` type has the following layout of information internally:

```
struct any {
    char *anyType; /* the typecode of the any */
    void *anyValue; /* the value stored in the any */
    char release_flag; /* whether the Orb or the user has control of Any */
};
```

`Any`s are opaque in PL/I, that is, they are referenced via a pointer to the `any` structure.

The following example based on the above IDL definition and generated PL/I data definitions illustrates access to the type and data of an `any`.

```
declare my_short fixed bin(15);
declare my_long fixed bin(31);

declare 1 example_temp_any_args aligned, 3 result type(any);

/* Provided and generated typecode name constants */
declare corba_type_short char(1) value('s');
declare corba_type_long char(1) value('l');
. . . /* other constants omitted */

declare example_type_code char(11);
```
call typeget(example_temp_any_args.result, 
    example_type_code, 
    length(example_type_code));
if example_type_code = CORBA_type_short then 
do;
call anyget(example_temp_any_args.result,addr(my_short));
display('my_short = ' || my_short);
End;
else if example_type_code = corba_type_long then 
do;
call anyget(example_temp,my_long,addr(my_long));
display('My_Long = ' || my_long);
end;

Changing the contents of the any requires setting the type code and then storing the 
new data.

my_short = 12;

call typeset(example_temp_any_args.result, 
    example_type_code, 
    length(example_type_code));
call anyset(example_temp_any_args,addr(my_short));

2.9 Mapping for Exceptions

An IDL exception maps to a PL/I structure and a character data item with a value that 
uniquely identifies the exception.

interface example {
    exception bad {
        long value1;
        string<32> reason;
    };
    
    exception worse {
        short value2;
        string<16> errorcode;
        string<32> reason;
    };
    
    void addName(in string name) raises(bad, worse);
}

maps to the following PL/I

declare exc_example_badchar(16) value('exc_example_bad ');
declare exc_example_worsechar(18) value('exc_example_worse ');
declare 1 example_user_exceptions,
3 d fixed bin(31),
3 u union,
  5 exception_bad,
    7 value1 fixed bin(31),
    7 reason char(32),
  5 exception_worse,
    7 value2 fixed bin(15),
    7 errorcode char(16),
    7 reason char(32);

The values shown for _exC_example_bad_ and _EXC_EXAMPLE_WORSE_ are examples only and are not intended to indicate the actual format of the unique exception identifiers.

The server signals an error to the client by using the **PODERR** function.

```pli
if name = '' then
do;
  exception_bad.reason = 'No name';
  exception_bad.value1 = 99999;
  call poderr(exc_example_bad);
end;
```

To test for errors, the client side shall be set up as follows.

```pli
CHECK_ERRORS: PROC(FUNCTION_NAME) RETURNS(FIXED BIN(31));
dcl exc_name char(64);
dcl exception_info char(64);
if exception_number = pod_user_exception then
do;
  display('example_user_exception');
  strget(exception_id,64,exc_name);
  select(exc_name);
    when('exc_example_bad')
      do;
        display('value1 = ' || example_bad.value1);
        display('reason = ' || example_bad.reason);
      end;
    when('exc_example_worse')
      do;
        display('value2 = ' || example_worse.value2);
        display('errorcode = ' || exmaple_worse.errorcode);
        display('reason = ' || example_worse.reason);
      end;
  end;
return_code=completion_status_no;
end;
else if exception_number ^= 0 then
do;
  strget(exception_text,64,exception_info);
  display('system_exception');
  display('exception_number = ' || exception_number);
  display('exception = ' || exception_info);
end;
```
return_code=completion_status_no;
end;
else
  return_code=completion_status_yes;

return(retrun_code);
END CHECK_ERRORS;
...

CALL PODSTAT(POD_STATUS_INFORMATION,
          ADDR(EXAMPLE_USER_EXCEPTIONS));
CALL PODREG(ADDR(EXAMPLE_INTERFACE));
IF CHECK_ERRORS('PODREG') ^= COMPLETION_STATUS_YES THEN RETURN;

2.10 Mapping for Interfaces

The use of an interface type in IDL denotes an object reference. Each interface referenced by an attribute or an operation shall be mapped to a POINTER, aliased to OBJECT for clarity. To give an example:

interface Account {
  // implementation omitted
  ...
};

interface Bank {
  attribute Account mainAccount;
  ...
  Account getAccount(in long accountID);
};

The following PL/I argument structure shall be generated for attribute mainAccount:

declare 1 Bank_mainAccount_attr,
         3 result         type(object);

The following structure shall be generated for the getAccount operation:

declare 1 Bank_getAccount_args,
         3 accountID      fixed bin(31),
         3 result         type(object);

2.11 Mapping for Typedefs

PL/I supports typedefs via the use of DEFINE ALIAS. For example,

typedef short myshort;

attribute myshort msh;
maps to

\begin{verbatim}
define alias myshort fixed bin(15);
dcl msh     type(myshort);
\end{verbatim}

### 2.12 Mapping for Valuetypes

An IDL valuetype is mapped to PL/I by a pointer to the valuetype data along with accessor procedures to get and retrieve data members, to invoke operations on the valuetype and also procedures for initializing and destroying the valuetype. Complementing these are four general valuetype procedures used to increment and decrement the valuetype reference count, duplicating the valuetype and returning the current reference count of the valuetype. PL/I structures used for setting / retrieving the public valuetype data members and for invoking operations are also supplied. Finally, a based structure is created that contains a member for holding the reference count and for each public and private data attribute. These are only used by the valuetype procedures themselves. Private members may not be accessed and no accessor procedures are therefore supplied. These procedures and structures are contained in a separate include file for use by the valuetype procedures.

The valuetype pointer stores the current valuetype information; that is, the reference count and the contents of all public and private data members. Each valuetype procedure takes two pointer arguments, one for the valuetype itself and a pointer to the structure containing the data associated with the valuetype data member or operation.

The mapping for the valuetype names is performed as follows:

The valuetype pointer is mapped to the IDL valuetype name and suffixed _VT.

The PL/I procedures names for the valuetype operations are the operation names prefixed by vt_ValuetypeName_. There is a ‘get’ and ‘set’ procedure for each data member and the procedure names are formed as set_ and get_ along with the valuetype member identifier (up to the 100 character name limit in PL/I).

The associated valuetype structures for accessing the data are mapped to the valuetype data member, suffixed _vtype for the type description and _vtattr for the data members. For valuetype operations, the structure is suffixed _vtype for the type description and _vtargs for the actual argument placeholder. This matches the suffixing-convention for interface attributes and operations.

#### 2.12.1 Valuetype Data Members

For each public valuetype members, a ‘get’ and ‘set’ operation is provided for accessing the contents of the contents of the valuetype. The outline of an example of this is shown below and in Section 2.12.4, “Valuetype Example,” on page 2-16 for a more complete example of the valuetype mapping in PL/I.
As PL/I does not support the concept of public and private data members, all data is stored within the PL/I valuetype pointer. This includes the reference count for the valuetype as well as the contents of the private and public data members for the valuetype. Each valuetype procedure takes a valuetype pointer as its first parameter for extracting / inserting data. The other parameter is used to get / set the data into the given valuetype and this is passed to the procedure as the address of the procedure’s data structure. This is illustrated in the example given below.

### 2.12.2 Valuetype Data Example

The outline to a PL/I version of a valuetype is given below. A more comprehensive example is shown in Section 2.12.4, “Valuetype Example,” on page 2-16, the version here is to put the above descriptions into context. The ‘\$include VALUEBASE;’ statement is generated for each valuetype and the contents of this include file is described under Section 2.12.5, “Generic Valuetype Procedures,” on page 2-19.

```pli
valuetype Val {
   public Val t;
   private long v;
   public string w;
};

/* PL/I */
\$include VALUEBASE;  /* location of general valuetype procs */

vt_Val_get_t: PROC(vtptr,p_vtargs);
...
END vt_Val_get_t;

vt.Val_set_t: PROC(vtptr,p_vtargs);
...

vt_Val_get_w: PROC(vtptr,p_vtargs);
...

vt_Val_set_w: PROC(vtptr,p_vtargs);
...

vt.Val_init: PROC(vtptr);
...

vt_Val_delete: PROC(vtptr);
...

dcl 1 Val_t_vtype based,
   3 result        ptr           init(sysnull());

dcl 1 Val_w_vtype based,
   3 result        type(string) init(sysnull());

dcl 1 Val_vtype based,
   3 ctl           ptr           init(sysnull());
   3 refct fixed bin(31) init(0),
```
2.12.3 Valuetype Operations

Valuetype operations are mapped in a similar fashion as interface operations with procedures matching each operation. An example of how a valuetype with operations gets mapped to PL/I is shown below in the Valuetype Example below. The general operations (such as incrementing a valuetype’s reference count) are located in the VALUEBASE include file. These operations are described in Section 2.12.5, “Generic Valuetype Procedures,” on page 2-19.

2.12.4 Valuetype Example

To give a better idea of how a valuetype is mapped to PL/I, the Example valuetype below is converted to PL/I and discussed in more detail. Some sample procedure implementation code is also shown.

// IDL
valuetype Example {
    short op1();
    long  op2(in Example x);
    private short val1;
    public long val2;
    private string val3;
    private float val4;
};

/* PL/I */
%include VALUEBASE;

vt_Example_op1: PROC(vtptr,p_vtargs);
    dcl vtptr ptr byvalue;
    dcl p_vtargs ptr byvalue;
    dcl vt based(vtptr) like Example_vtype;
    dcl vt_args based(p_vtargs)
like Example_op1_vtype;
   
   if vtptr=sysnull() || p_vtargs=null/* something wrong! */
   /* signal an error */;
   ...
END vt_Example_op1;

vt_Example_op2: PROC(vtptr,p_vtargs);
   dcl vtptr          ptr byvalue;
   dcl p_vtargs       ptr byvalue;
   dcl vt             based(vtptr)
                    like Example_vtype;
   dcl vt_args        based(p_vtargs)
                    like Example_op2_vtype;
   ...
END vt_Example_op2;

vt_Example_get_val2: PROC(vtptr,p_vtargs);
   dcl vtptr          ptr byvalue;
   dcl p_vtargs       ptr byvalue;
   dcl vt             based(vtptr)
                    like Example_vtype;
   dcl vt_args        based(p_vtargs)
                    like Example_val2_vtype;
   ...
END vt_Example_get_val2;

vt_Example_get_val2: PROC(vtptr,p_vtargs);
   dcl vtptr          ptr byvalue;
   dcl p_vtargs       ptr byvalue;
   dcl vt             based(vtptr)
                    like Example_vtype;
   dcl vt_args        based(p_vtargs)
                    like Example_val2_vtype;
   ...
END vt_Example_get_val2;

vt_Example_init: PROC(vtptr,p_vtargs);
   dcl vtptr          ptr byaddr;
   dcl p_vtargs       ptr byvalue;
   dcl vt             based(vtptr)
like Example_vtype;

dcl vt_args           based(p_vtargs)
like Example_op2_vtype;

if vtptr=sysnull() || p_vtargs=null/* something wrong! */
/* signal an error */;

alloc vt;
END vt_Example_init;

vt_Example_delete: PROC(vtptr,p_vtargs);
dcl vtptr           ptr byaddr;
dcl p_vtargs        ptr byvalue;
dcl vt              based(vtptr)
like Example_vtype;
dcl vt_args         based(p_vtargs)
like Example_op2_vtype;

if vtptr=sysnull() || p_vtargs=null/* something wrong! */
/* signal an error */;

vt.refct=1;
call vt_decref(vtptr);
END vt_Example_delete;

dcl 1 Example_op1_vtype based,
  3 result        fixed bin(31) init(0);

dcl 1 Example_op2_vtype based,
  3 x              ptr;
  3 result        fixed bin(31) init(0);

dcl 1 Example_val2_vtype based,
  3 result        fixed bin(31) init(0);


dcl 1 Example_vtype based,
  3 ctl           ptr           init(sysnull()),
  3 refct         fixed bin(31) init(0),
  3 val1          fixed bin(15) init(0),
  3 val2          fixed bin(31) init(0),
  3 val3          ptr           init(sysnull()),
  3 val4          float dec(6)  init(0);

dcl 1 Example_pepv_vtype based,
  3 Example_pepv  ptr           init(sysnull());

dcl 1 Example_epv_vtype based,
  3 ctl           ptr,
  3 op1           entry limited,
  3 op2           entry limited,
  3 get_val2      entry limited,
In each valuetype procedure, the valuetype pointer gets mapped onto the valuetype structure in order to access the various elements of the valuetype. This is similar to how the second argument is mapped in order to access the incoming type. vt_example_get_val2 shows how this is utilized. A check should always be made to make sure the valuetype pointer and data are both valid.

The generic layout is shown for each of the procedures and a very simple implementation is shown for val2 to show how the mapped pointers are used. Again, notice that there are only procedures defined for getting and setting the contents of val2. Since val1 is declared as being private, only the valuetype procedure has access to this data member. Val2 on the other hand is public and so procedures are defined to access its contents. Finally, the _init and _delete procedures are provided for initialising the valuetype for use and destroying the valuetype when it’s reference count reaches 0.

2.12.5 Generic Valuetype Procedures

There are four generic procedures that are available for every valuetype. They are defined as follows and shall be stored in the VALBASE include file. The implementation code for these procedures are generic.

vt_incref(ptr); /* IN: valuetype pointer */ /* increments the valuetype’s reference count */

vt_decref(ptr); /* IN: valuetype pointer */ /* decrements the valuetype’s reference count */

vt_duplicate(inptr,          /* IN : orig vt ptr  */
            outptr);        /* OUT: duplicate vt ptr  */ /* duplicates the valuetype */

vt_refcount(ptr,             /* IN : valuetype pointer */
            fixed bin(31));  /* OUT: reference count */ /* returns the reference count for the valuetype */

An example of some of their implementations follows. These examples are for information purposes only.

vt_incref: PROC(vtptr);
    dcl vtptr            ptr byvalue;
    dcl 1 vt              based(vtptr),
      3 ctl        ptr,
      3 refcount  fixed bin(31);
    vt.refcount=vt.refcount+1;
2.12.6 Value Boxes

Value boxes are mapped to PL/I in a similar fashion to normal operations. There are two available value box operations, a _boxed_get and a _boxed_set, both prefixed by the valuetype name. The _boxed_get is for valuetype content retrieval and the _boxed_set is for replacing the contents of the valuetype. Value boxes can be used for simple types only. These cover (un)signed integer types, boolean, octet, char, float, (long) doubles, enumerated types and string types. An example of how a value box could be used is shown below.

2.12.7 Value Box Example

An example of a string value box is shown below.

// IDL
valuetype StringValue string;
interface X { void op(out string s); };

END vt_incref;

vt_decref: PROC(vtptr);
  dcl vtptr ptr byvalue;
  dcl 1 vt based(vtptr),
    3 ctl ptr,
    3 refcount fixed bin(31);
  vt.refcount=vt.refcount-1;
  if vt.refcount=0 then
    /* destroy valuetype structure */;
  end vt_decref;

vt_refcount: PROC(vtptr,count);
  dcl 1 vtptr ptr byvalue;
  dcl count fixed bin(31) byaddr;
  dcl 1 vt based(vtptr),
    3 ctl ptr,
    3 refcount fixed bin(31);
  count=vt.refcount;
END vt_incref;

vt_duplicate: PROC(vtptr1,vtptr2);
  /* implementation omitted */
END vt_duplicate;
/* PL/I – mainline with a call to boxed_get */
dcl myString type(string);
dcl myStringVT like StringValue_vtype;
...
call vt_StringValue_boxed_get(myStringVT,addr(mystring));

/* PL/I include file of the Value Box procedures for StringValue */
#include VALUEBASE;

vt_StringValue_boxed_get: PROC(vtptr,p_vtargs);
dcl vtptr ptr byvalue;
dcl p_vtargs ptr byaddr;
dcl vt based(vtptr)
   like StringValue_vtype;
dcl vt_args based(p_vtargs)
   like StringValue_boxtype;
/* check for pointers' validity omitted */
call STRFREE(vt_args);
call STRDUPL(vt.result,vt_args);
END vt_StringValue_boxed_get;

vt_StringValue_boxed_set: PROC(vtptr,p_vtargs);
dcl vtptr ptr byvalue;
dcl p_vtargs ptr byvalue;
dcl vt based(vtptr)
   like StringValue_vtype;
dcl vt_args based(p_vtargs)
   like StringValue_boxtype;
/* check for pointers' validity omitted */
call STRFREE(vt.result);
call STRDUPL(vt_args,vt.result);
END vt_StringValue_boxed_set;

dcl 1 StringValue_vtype based,
   3 ctl ptr init(sysnull());
   3 pepv ptr init(sysnull()),
   3 refct fixed bin(31) init(0),
   3 result type(string) init(sysnull());

dcl 1 StringValue_pepv_vtype based,
   3 StringValue ptr init(sysnull());

dcl 1 StringValue_epv_vtype based,
   3 StringValue_boxed_get entry limited,
   3 StringValue_boxed_set entry limited;
dcl StringValue_boxtype ptr based;

dcl 1 vt_StringValue_epv like StringValue_epv_vtype;

2.12.8 Abstract Valuetypes

Abstract IDL valuetypes follow the same PL/I mapping rules as concrete IDL valuetypes, except that they do not have any data members. Also, as abstract valuetypes do not have state information, while the refct variable is present, it is not used and the `%include VALUEBASE;` statement, the `_init` and the `_delete` procedures are omitted as these all refer to a valuetype’s state information.

2.12.9 Valuetype Inheritance

Example:

```
// IDL
abstract valuetype A {
    void op();
};

valuetype B supports A {
    public short data;
};

/* PL/I */
%include VALUEBASE;

vt_A_op: PROC(vtptr,p_vtargs);
    dcl vtptr  ptr byvalue;
    dcl p_vtargs ptr byvalue;
    dcl vt    based(vtptr)
              like A_vtype;
    dcl vt_args based(p_vtargs)
              like A_op_type;

    /* implementation code here */
    ...
END vt_A_op;

vt_B_op: PROC(vtptr,p_vtargs); /* as B inherited A’s op */
    dcl vtptr  ptr byvalue;
    dcl p_vtargs ptr byvalue;
    dcl vt    based(vtptr)
              like B_vtype;
    dcl vt_args based(p_vtargs)
              like A_op_type; /* inherits A’s attributes */

    /* implementation code here */
```
The main point to note from the above code is the extra line in `B_pepv_vtype` to point to interface A’s operation procedures. The developer has the choice of either using the implementation of A’s operation as defined in interface A or a unique implementation for use with valuetype B.

For example, to set up valuetype B so that it calls A’s definition of procedure op:
/* epv initialization */
vt_A_epv.op = addr(vt_A_op);

vt_B_epv.get_data = vt_B_get_data;
vt_B_epv.set_data = vt_B_set_data;

/* pepv initialization */
vt_A_pepv.A_epv = vt_A_epv;
vt_B_pepv.A_epv = vt_A_epv;
vt_B_pepv.B_epv = vt_B_epv;

To set up valuetype B so that it calls B’s redefined procedure op (inherited from A) instead:

/* epvs and pepvs for operations */
dcl 1 vt_A_epv like A_epv_vtype;
dcl 1 vt_B_epv like B_epv_vtype;
dcl 1 vt_B_A_epv like A_epv_vtype; /* used for B-spec op impl */

dcl 1 vt_A_pepv like A_pepv_vtype;
dcl 1 vt_B_pepv like B_pepv_vtype;

/* epv initialization */
vt_A_epv.op = addr(vt_A_op);

vt_B_epv.get_data = vt_B_get_data;
vt_B_epv.set_data = vt_B_set_data;

vt_B_A_epv.op = addr(vt_B_op); /* B-specific implementation of op */

/* pepv initialization */
vt_A_pepv.A_epv = vt_A_epv;
vt_B_pepv.A_epv = vt_B_A_epv; /* choose B-specific epv */
vt_B_pepv.B_epv = vt_B_epv;

2.13 Portable Object Adapter Mapping

This section describes the details of the IDL-to-PL/I language mapping that apply to the Portable Object Adapter. It 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. Note that the default for the POA for the PL/I mapping is single-threaded as PL/I cannot be run multithreaded on all platforms.
As stated in the introduction to the PL/I mapping, if a mapping is to be implemented for an implementation of the PL/I language, which is single-threaded only, or cannot be guaranteed to be thread-safe, the POA ThreadPolicy shall only accept the value of SINGLE_THREAD_MODEL. Other values shall raise the InvalidPolicy exception. If the underlying PL/I implementation is known to be thread-safe or multi-threaded, the POA ThreadPolicy shall have the standard default of ORB_CTRL_MODEL. Note that all auxiliary functions must be implemented by thread-safe code in this case.

2.13.1 PortableServer Procedures

Objects that are registered with POAs use sequences of octet as object identifiers, specifically the PortableServer::POA::ObjectId type. There are two PL/I procedures for manipulating these ObjectIds, namely POS_GET_OBJECTID and POS_SET_OBJECTID. These convert the ObjectId to a PL/I string and vice versa, respectively. These are described in more detail below and shall be stored in the CORBA include file. Note that all PortableServer-specific procedures are prefixed POS_.

POS_GET_OBJECTID(PTR BYVALUE, /* POA::ObjectId */ CHAR(*) BYADDR, /* PL/I string */ FIXED BIN(31) BYVALUE); /* Length of PL/I str */ /* Extracts the ObjectId into the given character string */

POS_SET_OBJECTID(CHAR(*) BYVALUE, /* Blank-terminated PL/I string */ PTR BYADDR); /* POA::ObjectId to be set*/ /* Sets an ObjectId using the given character string */

If the string supplied to POS_GET_OBJECTID is too long for the ObjectId string, a MARSHALL::LENGTH_TOO_LARGE exception shall be thrown. Also, if an invalid ObjectId is passed to POD_GET_OBJECTID, a CORBA OBJECT_NOT_EXIST::OBJECT_NOT_FOUND exception shall be thrown.

2.13.2 Mapping for PortableServer::ServantLocator::Cookie

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

define cookie alias ptr;

For the PL/I 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:

POS_SERVLOC_PREINVOKE(PTR BYVALUE, /* Object ID */ LIMITED ENTRY, /* PortableServer POA */ CHAR(*) BYVALUE, /* CORBA ID */ FIXED BIN(31) BYVALUE, /* Length of CORBA ID */ PTR BYADDR); /* Ptr to Cookie */ /* Equivalent to PortableServer::ServantLocator::preinvoke() */
POS_SERVLOC_POSTINVOKE(PTR BYVALUE, /* Object ID */
LIMITED ENTRY BYVALUE, /* PortableServer POA */
CHAR(*) BYVALUE, /* CORBA ID */
FIXED BIN(31) BYVALUE, /* Length of CORBA ID */
PTR BYVALUE, /* Cookie */
PTR BYVALUE); /* Servant */
/* Equivalent to PortableServer::ServantLocator::postinvoke() */

2.13.3 PortableServer::Servant Mapping

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

The PortableServer::Servant type maps into PL/I as follows:

define pos_servant alias ptr;

Associated with a servant is a table of pointers to method procedures. 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 pos_servant is defined as follows:

dcl 1 pos_servantbase_epv based,
    3 ctl   ptr,
    3 finalize limited entry,
    3 default_poa limited entry;

The two limited entry procedures are defined as follows.

FINALIZE(PTR BYADDR); /* Servant */
// Finalize definition

DEFAULT_POA(PTR BYADDR, /* Servant */
    LIMITED ENTRY); /* Portable POA */
// Default POA definition

The pos_servantbase_epv’s ctl 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 shall be shared among multiple servants, this
member is not suitable for per-servant data. The second member is a pointer to the
finalization procedure for the servant, which is invoked when the servant is
etherealized. The other procedure pointers correspond to the usual servant operations.

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

/* PEPV is a pointer to the EPV */
dcl pos_servantbase_pepv   ptr based;

dcl 1 pos_servantbase based,
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 pos_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, pepv is standardized to allow portable access through it.

2.13.4 Interface Skeletons

All PL/I 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 (entry point vectors) 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):

```
dcl 1 poa.Counter based,
  3 ctl ptr,
  3 pepv type(pos_servantbase_pepv);
```

As with pos_servantbase defined in the PortableServer::Servant mapping above, the name of the second member is standardized to pepv (pointer to the entry point vector) for portability. The EPV-generated for the skeleton is a bit more interesting. For the Counter interface defined above, it appears as follows:

```
dcl 1 poa.Counter_epv based,
  3 ctl ptr,
  3 add limited entry;
```

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

```
dcl 1 poa.Counter_pepv based,
  3 base_epv ptr,
  3 Counter_epv ptr;
```
The first member of the `poa_counter_pepv` struct is a pointer to the `pos_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_Countepv`.

The pointers to EPVs in the PEPV 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 `pos_servantbase`. For example, consider the following complicated interface hierarchy:

```idl
// 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 PEPV structure for interface `G` is generated as follows:

```pli
/* PL/I */
dcl 1 poa_G_epv based,  
    3 ctl ptr,  
    3 foo limited entry;

dcl 1 poa_G_pepv based,  
    3 base_epv ptr,  
    3 A_epv ptr,  
    3 B_epv ptr,  
    3 C_epv ptr,  
    3 D_epv ptr,  
    3 E_epv ptr,  
    3 F_epv ptr,  
    3 G_epv ptr;
```

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.

### 2.13.5 Servant Structure Initialization

Each servant requires initialization and etherealization, or finalization, procedures. For `pos_servantbase`, the ORB implementation shall provide the following procedures:

```c
POS_SERVANTBASE_INIT(PTR BYADDR); /* Servant */  
// PortableServer::ServantBase initializer
```
These procedures are named by appending _INIT and _FINI to the name of the servant, respectively.

The argument to the init procedure shall be a valid pos_servant whose pepv member has already been initialized to point to a PEPV structure. The initialization procedure, POS_SERVANTBASE_INIT, shall perform ORB-specific initialization of the pos_servantbase and shall initialize the finalize struct member of the pointed-to pos_servantbase_epv to point to the POS_SERVANTBASE_FINI procedure 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 pos_servantbase_epv structure is NULL when the POS_SERVANTBASE_INIT procedure is called, its value is set to point to the DEFAULT_POA procedure, which returns an object reference to the root POA.

If a servant pointed to by the pos_servant passed to an initialization procedure has a NULL pepv member, or if the pos_servant argument itself is NULL, no initialization of the servant is performed, and the CORBA::BAD_PARAM standard exception is thrown. This also applies to interface-specific initialization procedures, which are described below. The finalization procedures only cleans up ORB-specific private data. It is the default finalization procedure 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 finalize procedure for a given servant by initializing the pos_servantbase_epv finalize pointer with a pointer to a finalization procedure made specifically for that servant; however, any such overriding procedure must always ensure that the POS_SERVANTBASE_FINI procedure is invoked for that servant as part of its implementation. The results of a finalization procedure failing to invoke POS_SERVANTBASE_FINI are implementation-specific, but may include memory leaks or faults that could crash the application.

If a servant passed to a finalization procedure has a NULL EPV member, or if the pos_servant argument itself is NULL, no finalization of the servant is performed, and the CORBA::BAD_PARAM standard exception is raised. This also applies to interface-specific finalization procedures, which are described below.

Normally, the POS_SERVANTBASE_INIT and POS_SERVANTBASE_FINI procedures are not invoked directly by applications, but rather by interface-specific initialization and finalization procedures generated by an IDL compiler. For example, the initialization and finalization procedures generated for the Counter skeleton are defined as follows.

```
POA_COUNTER_INIT: PROC (POA_COUNTER_PTR);

dcl poa_counter_ptr  ptr byaddr;

/* First call the immediate base interface init */
```
/* procs in the left-to-right order of inheritance */
call POS_SERVANTBASE_INIT(poa_counter_ptr);

/* Now perform poa_counter initialization */
...

END POA_COUNTER_INIT;

POA_COUNTER_FINI: PROC(POA_COUNTER);

dcl poa_counter ptr byaddr;

/* First perform poa_counter cleanup */
...

/* Then call immediate base interface fini procs */
/* in the right-to-left order of inheritance */
call POS_SERVANTBASE_FINI(poa_counter);

END POA_COUNTER_FINI;

The procedure names are defined as follows. The interface name is prefixed POA_ and suffixed with _INIT for the initialization procedure and _FINI for the finalization procedure. The address of a servant shall be passed to the initialization procedure 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 initialization procedure before registering it or allowing it to be activated are implementation-specific.

2.13.6 Application Servants

It is expected that applications should 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:

dcl 1 appservant based,
    3 base    type(poa_counter),
    3 ctrvalue fixed bin(31);

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

SRVADD: PROC(APPSRV, INLONG, OUTLONG);

%include SRVTYPE;

dcl appsrv like appservant byvalue;
dcl inlong fixed bin(31) byvalue;
The application could initialize the servant dynamically as follows.

```pli

dcl outlong       fixed bin(31)   byaddr;
...

appsrv.ctrvalue=appsrv.ctrvalue+inlong;
outlong=appsrv.ctrvalue;

END SRVADD;
```

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

```pli

call poa_Counter_init(my_servant);
```

If the application requires a special destruction procedure for `my_servant`, it shall set the value of the `pos_servantbase_epv`'s `finalize` member either before or after calling `poa_Counter_init`:

```pli

base_epv.finalize=proc_my_finalizer;
```
Note that if the application statically initialized the `finalize` member before calling the servant initialization procedure, explicit assignment to the `finalize` member as shown here is not necessary, since the `POS_SERVANTBASE_INIT` procedure shall not modify it if it is non-NULL.

### 2.13.7 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);
};
```

then the parameter structure for the `op5` argument must be defined as follows.

```
dcl 1 op5_type based,
   3 servant      type(pos_servant) init(sysnull()),
   3 arg6         fixed bin(31)     init(0),
   3 result       fixed bin(31)     init(0);
```

The `servant` member (which is an instance of `pos_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 `servant` member must be the first member of the structure.

### 2.14 Miscellaneous Other Mappings (Generator Optional)

CORBA simple types for setting `ANY`: typecodes defined as `VALUE`-initialized character strings.

These simple types are defined as follows.

- `DCL CORBA_TYPE_ANY         CHAR(01)  VALUE('a');`
- `DCL CORBA_TYPE_BOOLEAN     CHAR(01)  VALUE('b');`
- `DCL CORBA_TYPE_CHAR        CHAR(01)  VALUE('c');`
- `DCL CORBA_TYPE_DOUBLE      CHAR(01)  VALUE('d');`
- `DCL CORBA_TYPE_FLOAT       CHAR(01)  VALUE('f');`
- `DCL CORBA_TYPE_LONG        CHAR(01)  VALUE('l');`
- `DCL CORBA_TYPE_OCTET       CHAR(01)  VALUE('o');`
- `DCL CORBA_TYPE_SHORT       CHAR(01)  VALUE('s');`
- `DCL CORBA_TYPE_ULONG       CHAR(02)  VALUE('ul');`
- `DCL CORBA_TYPE_USHORT      CHAR(02)  VALUE('us');`
- `DCL CORBA_TYPE_USTRING     CHAR(01)  VALUE('0');`

Generated typecodes: also `VALUE`-initialized character strings.

Padding for PL/I structures: identifier: *

All PL/I identifiers are initialized as follows; that is, numeric, strings, and pointers:
default (real) \hspace{5mm} \text{init}((*)0); \\
default (pointer) \hspace{5mm} \text{init}((*)\text{sysnull}()); \\
default (varying | \hspace{1mm} \text{nonvarying} | \hspace{1mm} \text{varyingz}) \hspace{5mm} \text{init}((*)''); \\

Other CORBA types not mentioned before:

\text{CORBA-Object maps to} \hspace{5mm} \text{TYPE(OBJECT) aliased to a POINTER.}

2.15 Memory Handling by the POD

The charts below detail who is responsible for the allocation and de-allocation of memory at the various stages of a POD application.
2.15.1 Unbounded Sequences

<table>
<thead>
<tr>
<th>Unbounded Sequence</th>
<th>Client Application</th>
<th>PODEXEC</th>
<th>PODGET</th>
<th>Server Application</th>
<th>PODPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>1. SEQALOC</td>
<td></td>
<td>(allocs memory)</td>
<td>5. (read)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. (write)</td>
<td></td>
<td>4. (receive)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. (send)</td>
<td></td>
<td></td>
<td>(frees memory)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6. SEQFREE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inout</td>
<td>1. SEQALOC</td>
<td></td>
<td>(allocs memory)</td>
<td>5. (read)</td>
<td>7. (send)</td>
</tr>
<tr>
<td></td>
<td>2. (write)</td>
<td></td>
<td>4. (receive)</td>
<td>(frees memory)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. (send)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8. (receive)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>9. (read)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10. SEQFREE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Out / Return</td>
<td>5. (read)</td>
<td></td>
<td>(allocs memory)</td>
<td></td>
<td>(POD frees</td>
</tr>
<tr>
<td></td>
<td>6. SEQFREE</td>
<td></td>
<td>4. (receive)</td>
<td>mem. alloc’d by</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SEQALOC)</td>
<td></td>
</tr>
</tbody>
</table>

2.15.2 Unbounded Sequences and Memory Management

Sequences passed as OUT parameters/result are created with SEQALOC. These are owned by the POD once PODPUT is called and shall be automatically SEQFREEd on the server side once their contents have been sent back to the client. The client program must free the sequence received from an OUT parameter. The memory management of all other sequences (for example, temporary sequences) created with SEQALOC is the responsibility of the programmer. When the sequence is no longer required it should be deallocated using the SEQFREE function.
**INOUT sequences**

If the content of the sequence is modified (by replacing the pointer with a new sequence allocated via `SEQSET`), then the POD shall (correctly) FREE both the original IN sequence and the replaced OUT sequence. If the content remains unchanged, then only the original sequence should be `SEQFREE`d.

### 2.15.3 Unbounded Strings

<table>
<thead>
<tr>
<th>Unbounded Strings</th>
<th>In</th>
<th>Client</th>
<th>PODEXEC</th>
<th>PODGET</th>
<th>Server</th>
<th>Application</th>
<th>PODPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. STRSET</td>
<td></td>
<td>Application</td>
<td>PODEXEC</td>
<td>PODGET</td>
<td>Application</td>
<td>PODGET</td>
<td>Application</td>
</tr>
<tr>
<td>2. (send)</td>
<td></td>
<td>(allocs memory)</td>
<td></td>
<td></td>
<td>4. STRGET</td>
<td>(frees memory)</td>
<td></td>
</tr>
<tr>
<td>3. (receive)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. MEMFREE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inout</th>
<th>1. STRSET</th>
<th></th>
<th>Application</th>
<th>PODEXEC</th>
<th>PODGET</th>
<th>Application</th>
<th>PODPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. (send)</td>
<td></td>
<td>(allocs memory)</td>
<td></td>
<td></td>
<td>4. STRGET-in</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. (receive)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5. MEMFREE-in</td>
<td>6. STRSET-out</td>
<td></td>
</tr>
<tr>
<td>9. STRGET</td>
<td>8. (receive)</td>
<td></td>
<td></td>
<td></td>
<td>7. (send)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10. MEMFREE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(frees memory)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Out / Return</th>
<th>1. STRSET</th>
<th></th>
<th>Application</th>
<th>PODEXEC</th>
<th>PODGET</th>
<th>Application</th>
<th>PODPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. STRGET</td>
<td>3. (receive)</td>
<td></td>
<td>(allocs memory)</td>
<td></td>
<td></td>
<td>(POD frees the memory allocated by STRSET)</td>
<td></td>
</tr>
<tr>
<td>5. MEMFREE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2.15.4 Unbounded strings and memory management

Strings passed as OUT parameters/result are created with `STRSET`. These are owned by the POD once `PODPUT` is called and shall be automatically `MEMFREE`d on the server side once their contents have been sent back to the client. `STRLENG` is used to find the length of a given string. It is up to the client program to free the string received via an OUT parameter.
The memory management of all other strings created with `STRSET` is the responsibility of the programmer. When the string is no longer required it should be deallocated using the `MEMFREE` auxiliary function.

**INOUT strings**

If the content of the string is modified (by replacing the pointer with a new string allocated via `STRSET`) the POD shall (correctly) FREE both the original IN string and the replaced OUT string. If the content remains unchanged, then only the original string should be FREEd.

### 2.15.5 In Summary

Memory handling must be done when using dynamic structures such as ‘Unbounded Sequences’ and ‘Unbounded Strings.’

On the client side, memory allocation and release for these dynamic structures must almost be done by the application itself, except for OUT-arguments. In this case allocation must therefore be done by the PODEXEC function to provide the result back into the application. Memory release for IN- and INOUT arguments is done by the client. Memory release for OUT arguments is taken care of by the POD itself for the server program. However, the client program must explicitly free the arguments received via an OUT parameter.

On the server side, memory release is always done by the POD for arguments passed via parameters.

### 2.16 POD Function Summary

The following table summarizes the functions that are defined in the “PL/I object adapter,” in pseudo code. An explanation of how to use each function follows with an example of how to call it from PL/I. Utility functions for memory management are also described.

The appendix describes the general format of `CHECK_ERRORS`, an error-testing function that is used throughout this document but not actually part of the PL/I object adapter.

```c
ANYFREE(PTR); /* IN: anyInfoBlock */
// Frees memory allocated to an any

ANYGET(PTR,
  PTR); /* IN: anyInfoBlock */
// Extracts data out of an ANY

ANYSET(PTR,
  PTR); /* IN: anyInfoBlock */
// Inserts data into an ANY
```
MEMALLOC(PTR, /* OUT: pointer to memblock */
     FIXED BIN(31)); /* IN : amount of mem req'd */
// Allocates memory

MEMFREE(PTR); /* IN: pointer to memblock */
// Frees memory

OBJ2STR(PTR, /* IN : object reference */
     CHAR(*), /* OUT: IOR reference */
     FIXED BIN(31)); /* IN : IOR reference length */
// Returns an interoperable object reference (IOR)

OBJGTID(PTR, /* IN : object reference */
     CHAR(*), /* OUT: object ID string */
     FIXED BIN(31)); /* IN : string length */
// Retrieves the object ID from an IOR

OBJNEW(CHAR(*), /* IN : server name */
     CHAR(*), /* IN : interface name */
     CHAR(*), /* IN : object ID */
     PTR); /* OUT: object reference */
// Creates a unique object reference

OBJRIR(PTR, /* IN : object_ref */
     CHAR(*)); /* IN : desired_service */
// Returns an object reference to an object through which a service such as the Naming Service can be used

ORBARGS(CHAR(*), /* IN : arg string */
     FIXED BIN(31), /* IN : arg string len */
     CHAR(*), /* IN : ORB name */
     FIXED BIN(31)); /* IN : ORB name len */
// Initializes a client or server's connection to an ORB

MEMDBG(PTR, /* IN : pointer to memory */
     FIXED BIN(15), /* IN : size of memory dump */
     CHAR(*), /* IN : explanatory text str */
     FIXED BIN(15)); /* IN : len of text string */
// Output a formatted memory dump for the specified block of memory

PODERR(CHAR(*), /* IN: exception string */
     PTR); /* IN: addr(exception_buf)*/
// Signals a user exception to the ORB

PODEXEC(PTR, /* IN : object reference */
     CHAR(*), /* IN : operation name */
     PTR, /* INOUT: addr(op_buffer) */
     PTR); /* OUT : addr(user_exc_blk)*/
// Invokes an operation on the object
PODGET(PTR); /* INOUT: addr(opArgBuffer) */
// Gets IN and INOUT values

PODINIT(CHAR(*), /* IN: server name */
FIXED BIN(31)); /* IN: server name's len */
// Equivalent to impl_is_ready

PODINFO(PTR); /* OUT: status info pointer */
// Retrieves address of the status_information_buffer

PODPUT(PTR); /* INOUT: addr(opParamBuf) */
// Returns INOUT, OUT & result values

PODREG(PTR); /* IN: interface description */
// Describes an interface to the PL/I adapter

PODREGI(PTR, /* IN: interface description*/
PTR); /* OUT: object reference */
// Describes an interface to the PL/I adapter, returning an IOR

PODREQ(1, 3 PTR, /* IN: request info buffer */
3 PTR,
3 PTR,
3 PTR);
// Provides current request info

PODSRVR(CHAR(*), /* IN: server name */
FIXED BIN(31)); /* IN: server name length */
// Creates the server POA

PODSTAT(PTR); /* IN: status desc pointer */
// Registers status information buffer

SEQALOC(PTR, /* OUT: sequence ctrl data */
FIXED BIN(31), /* IN: length */
CHAR(*), /* IN: sequence typecode */
FIXED BIN(31)); /* IN: typecode length */
// Allocates storage for an unbounded sequence

SEQDUPL(PTR, /* IN: sequence data ptr */
PTR); /* OUT: dupl seq data ptr */
// Duplicates an unbounded sequence control block

SEQFREE(PTR); /* IN: sequence data ptr */
// Frees an unbounded sequence

SEQGET(PTR, /* IN: sequence data ptr */
FIXED BIN(31), /* IN: element number */
PTR); /* OUT: addr(seq_buffer) */
// Retrieves the element_number element of an unbounded sequence
SEQINIT(PTR, /* OUT: sequence data ptr */
   CHAR(*), /* IN: sequence typecode */
   FIXED BIN(31)); /* IN: typecode length */

   // Initializes a bounded sequence

SEQLEN(PTR, /* IN: sequence data ptr */
   FIXED BIN(31)); /* OUT: length of sequence */

   // Retrieves the current length of the sequence

SEQLSET(PTR, /* IN: sequence data ptr */
   FIXED BIN(31)); /* IN: new length of seq */

   // Changes the number of elements in the sequence

SEQMAX(PTR, /* IN: sequence data ptr */
   FIXED BIN(31)); /* OUT: max len of sequence */

   // Returns the maximum set length of the sequence

SEQSET(PTR, /* IN: sequence data ptr */
   FIXED BIN(31), /* IN: element number */
   PTR); /* IN: addr(seq_buffer) */

   // Stores the data into the element_number element of
   // an unbounded sequence.

STR2OBJ(PTR, /* IN: IOR string (null-terminated) */
   PTR); /* OUT: object reference */

   // Creates an object reference from an IOR

STRCON(PTR, /* INOUT: wide string #1 */
   PTR); /* IN: wide string #2 */

STRDUPL(PTR, /* IN: string pointer */
   PTR); /* OUT: duplicate string ptr */

   // Duplicates a given unbounded string

STRFREE(PTR); /* IN: unbounded string ptr */

   // Frees the storage of an unbounded string

STRGET(PTR, /* IN: string pointer */
   CHAR(*), /* OUT: PL/I string */
   FIXED BIN(31)); /* IN: PL/I string length */

   // Copies the contents of a dynamic string to a CHAR(n) data item

STRLENG(PTR /* IN: string pointer */
   FIXED BIN(31)); /* OUT: len of string */

   // Returns the actual length of an unbounded string

STRSET (PTR, /* OUT: string ptr - no pad */
   CHAR(*), /* IN: PL/I string */
   FIXED BIN(31)); /* IN: PL/I string length */
STRSETS(PTR,           /* OUT: string ptr with pad */
  CHAR(*),          /* IN : PL/I string */
  FIXED BIN(31));  /* IN : PL/I string length */
// Creates a dynamic string from a CHAR(n) data item

TYPEGET(PTR,        /* IN : addr(anyInfoBlock) */
  CHAR(*),        /* OUT: typecode */
  FIXED BIN(31)); /* IN : typecode length */
// Extracts the type of an ANY

TYPESET(PTR,        /* IN: addr(anyInfoBlock) */
  CHAR(*),        /* IN: typecode */
  FIXED BIN(31)); /* IN: typecode length */
// Inserts the type of an ANY

WSTRCON(PTR,        /* INOUT : wide string #1 */
    PTR);        /* IN    : wide string #2 */
// Concatenates two unbounded wide strings

WSTRDUP(PTR,        /* IN : wide string pointer */
    PTR);        /* OUT: duplicate string ptr */
// Duplicates a given unbounded wide string

WSTRFRE(PTR);  /* IN: unbounded w-string ptr*/
// Frees the storage of an unbounded wide string

WSTRGET(PTR,        /* IN : string pointer */
  CHAR(*),        /* OUT: PL/I string */
  FIXED BIN(31)); /* IN : PL/I string length */
// Copies the contents of an unbounded wide string to a
// CHAR(n) data item

WSTRLEN(PTR,        /* IN : wide string pointer */
  FIXED BIN(31));  /* OUT: len of wide string */
// Returns the length of an unbounded wide string

WSTRSET(PTR,        /* OUT: wstring ptr - no pad */
  CHAR(*),        /* IN : PL/I wide string */
  FIXED BIN(31)); /* IN : PL/I w-string length */

WSTRSTS(PTR,        /* OUT: string ptr with pad */
  CHAR(*),        /* IN : PL/I wide string */
  FIXED BIN(31)); /* IN : PL/I w-string length */
// Creates an unbounded wide string from a WIDECCHAR(n) data
// item

APPENDIX

CHECK_ERRORS(CHAR(*)) RETURNS(FIXED BIN(31));/* IN: procedure called */
In order for a client to invoke an object, it needs to do the following sequence of calls:

1. Call PODSTAT to register the POD_STATUS_INFORMATION. This shall enable the retrieval of the ORB’s status information for each call to the ORB.

2. Call ORBARGS to initialize a global ORB for the client. This function shall work in a similar fashion to the CORBA ORB_init call and must be called after PODSTAT.

3. Call PODREG to register each interface with the ORB. This shall provide the information required about the interface when invoking an operation on it.

4. Read in the IOR written by each server and call STR2OBJ on each to create an object reference. These object references shall then be used for the invocations.

5. The client is now ready to invoke an object. For each invocation set up the associated attribute or operation structure associated with the attribute / operation and pass in the associated object reference, the name of the operation, the address of the interface structure (as used in the PODREG call) and the associated user exception structure (a dummy structure shall be provided for operations without user exceptions).

The following example shows an implementation of the above steps using the interface described below.

```pli
interface client {
    attribute string<80> my_client_attribute;
    string<80> my_client_op(in long invalue);
};

CLIENT: PROC(IN ORB_ARG_STRING) OPTIONS(MAIN NOEXECOPS);

dcl in_orb_arg_string char(80) varying;
dcl orb_arg_string char(80) init(in_orb_arg_string);
dcl orb_name char(20) init('my_orb');
dcl operation char(256) init('');
...
dcl client_obj ptr;

/* The following include contains general setup info, eg declarations */
/* for pod_status_information, common types like CORBA_TYPE_SHORT */
%include CORBA;

%include CLIENTM; /* contains attr/op argument structures */
%include CLIENTR; /* contains CHECK_ERRORS function */
%include CLIENTX; /* contains the interface descript. for PODREG/EXEC*/
```
alloc pod_status_information set pod_status_ptr;

call podstat(pod_status_ptr); /* enable ORB status information */
if check_errors('podstat') ^= completion_status_yes then return;

call orbargs(orb_arg_string, length(orb_arg_string),
             orb_name, length(orb_name)); /* call ORB_init */
if check_errors('orbargs') ^= completion_status_yes then return;

call podreg(addr(client_interface));
if check_errors('podreg') ^= completion_status_yes then return;

/* The reading of the is omitted, storing IOR in client_ior */
...

call str2obj(client_ior, client_obj; /* create an object ref. */
if check_errors('str2obj') ^= completion_status_yes then return;

/* Examples of invoking the client object follow* /

/* Set the contents of an attribute called my_client_attribute */
operation=get_client_my_client_attribute;
client_my_client_attribute_attr.result='Send in data';

call podexec(client_obj,operation,
             addr(client_my_client_attribute_attr),
             addr(dummy_user_exc_block));
if check_errors('podexec') ^= completion_status_yes then return;

/* Set the contents of an operation called my_client_op */
operation=my_client_op;
client_my_client_args.invalue=46;

call podexec(client_obj,operation,
             addr(client_my_client_args),
             addr(dummy_user_exc_block));

if check_errors('podexec') ^= completion_status_yes then return;

put skip list('Return value from my_client_op =',
               client_my_client_args.result);

END CLIENT;

2.16.2 Server Implementation

There shall be three parts to the server implementation:

1. Initialization of the server – registering each interface the server is to support.
2. Dispatch – The entry point that is called to handle the interfaces registered above.
3. The Operation Implementation – How each operation is implemented.
Each part shall be in a separate PL/I module. Step 2 shall call the implementation in step 3. Common code for each interface (for example, the interface operation structures) shall be stored in include files accessible to all three PL/I modules described above. Step 1 shall be the server mainline; all three modules shall be compiled into one PL/I program.

The implementation of each step is described in more detail below.

### 2.16.2.1 Initialization of the Server

The following steps are required when a server is started. The following steps shall be stored in a standalone PL/I module.

1. Call **PODSTAT** to register the **POD_STATUS_INFORMATION**. This shall enable the retrieval of the ORB’s status information for each call to the ORB.

2. Call **ORBARGS** to initialize a global ORB for the server. This function shall work in a similar fashion to the **CORBA ORB_init** call and must be called after **PODSTAT**.

3. Call **PODREGI** for each interface to be registered, passing in the address of each interface description structure and a pointer to point at the returned IOR.

4. Call **OBJ2STR** passing in each object reference, and a character string to write the IOR string out to.

5. Write out each IOR.

6. Call **PODINIT** to inform the ORB that the server is now ready to accept requests.

The following example code, outlines the steps above and is based upon the interface described in the example under Object Invocation.

```pli
CLIENTV: PROC(IN_ORB_ARG_STRING) OPTIONS(MAIN NOEXECOPS);
dcl in_orb_arg_string char(80) varying;
dcl orb_arg_string char(80) init(in_orb_arg_string);
dcl orb_name char(20) init('my_orb');
...
dcl client_obj ptr;
/* following includes are described in 'Object Invocation' above */
%include CORBA;
%include CLIENTM;
%include CLIENTX;
%include CLIENTR;
alloc pod_status_information set(pod_status_ptr);
call podstat(pod_status_ptr);
if check_errors('podstat') ^= completion_status_yes then return;
call orbargs(orb_arg_string, length(orb_arg_string),
orb_name, length(orb_name)); /* call ORB_init */
if check_errors('orbargs') ^= completion_status_yes then return;
```
2.16.2.2 Server Dispatch

The server dispatch shall be set up as follows. The following shall be in a standalone PL/I module but shall call the server operation implementation PL/I module.

1. Declare an entry point for **DISPTCH**. This shall allow the POD to link to the server program and shall be the starting point when a server is invoked by a client, subsequent to its **PODINIT** call.

2. Declare the server implementation PL/I module as an external entry.

3. Call **PODREQ**, passing in the **REQUEST_INFO**. When a server is invoked by a client, the invocation request information shall be retrieved from this function.

4. Retrieve the interface name and operation name via calls to **STRGET**.

5. Call the server’s operation implementation PL/I module, passing in the interface and operation names.

The following example code segment describes the above steps.

```
ALIGNZ: PROC;

/* The following line enables the POD to link into this procedure */
DISPTCH: ENTRY;

dcl operation char(256);
dcl operation_length fixed bin(31);
/* the following is where the op’s implementations are held */
dcl CLIENTI ext entry(char(*),char(*));

/* following includes are described in ‘Object Invocation’ above */
%include CORBA;
%include CLIENTT;
%include CLIENTR;
```
call podreq(request_info);
if check_errors('podreq') ^= completion_status_yes then return;

/* Retrieve the interface name from the request information */
call strget(interface_name,
        interface,
        interface_length);
if check_errors('strget') ^= completion_status_yes then return;

/* Retrieve the operation name from the request information */
call strget(operation_name,
        operation,
        operation_length);
if check_errors('strget') ^= completion_status_yes then return;

/* call the implementation module to execute the server code for */
/* the retrieved attribute / operation                           */
call ALIGNI(interface,operation);

END ALIGNZ;

2.16.2.3 Server Operation Implementation Module

This section contains the actions to be performed for each attribute and operation on
the interface(s) associated with the server. The operation name and interface are passed
in by the server dispatch module. The following steps are required to perform each
operation.

1. Using a SELECT clause with the attribute / operation and interface names, call the
appropriate attribute / operation sub-procedure after calling PODGET with the
appropriate argument structure for the attribute / operation. This shall fill the IN
and INOUT values of the associated structure.

2. After the call completes, call PODPUT operation to marshal out the OUT and return
values.

The following example code segment (based on the IDL shown in the example for
‘Object Invocation’ above) demonstrates the above steps.

CLIENTI: PROC(INTERFACE,OPERATION);

dcl interface char(*);
dcl operation char(*);

%include CORBA;
%include CLIENTR;

...

select(interface);
    when('client') do;
        select(operation);
            when (get_client_my_client_attribute) do;
call podget(addr(client_my_client_attribute_attr));
if check_errors('podget') ^= completion_status_yes then return;

call proc_get_client_my_client_attribute
    (client_my_client_attribute_attr));

call podput(addr(client_my_client_attribute_attr));
if check_errors('podput') ^= completion_status_yes then return;
end;

/* similar again for setting my_client_attribute, omitted */
/* similar again for calling my_client_op, omitted for brevity */
otherwise do;
    put skip list('Invalid operation called!');
    return;
end; /*when*/
end; /*select*/
end; /*when*/
otherwise do;
    put skip list('No operations are defined for interface',
        interface);
    return;
end; /*otherwise*/
end; /*select*/

/* for brevity, only the implementation for operation my_client_op */
/* is shown below */
proc_client_my_client_op: PROC;
    put skip list('Contents of in param = ',
        client_my_client_op_args.invalue);
    client_my_client_op_args.result='Here is a result string';
END proc_client_my_client_op;
...

END CLIENTI;

2.16.3 ANYFREE

Summary

ANYFREE(PTR);       /* IN: anyInfoBlock */
// Frees memory allocated to an any

Description

ANYFREE is used to release memory from an ANY.

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

DCL 1 EXAMPL_TEMP_ANY_ARGS ALIGNED,
    3 RESULT PTR;

...  

CALL ANYGET (EXAMPL_TEMP_ANY_ARGS.RESULT, ADDR (WS_SHORT));

...  

CALL ANYFREE (EXAMPL_TEMP_ANY_ARGS.RESULT);

2.16.4 ANYGET

Summary

ANYGET(PTR, /* IN: anyInfoBlock */
    PTR); /* IN: addr(anyData) */
// Extracts data out of an ANY

Description

The ANYGET utility 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 can be used to extract the type of the ANY.

Note that it is the address of anyData that is passed to ANYGET.

Example

DCL 1 EXAMPL_TEMP_ANY_ARGS ALIGNED,
    3 RESULT PTR;

DCL WS_SHORT  FIXED BIN(15);
DCL WS_LONG   FIXED BIN(31);

...  

CALL TYPEGET (EXAMPL_TEMP_ANY_ARGS.RESULT, EXAMPLE_TYPE_CODE, 1);

SELECT (EXAMPLE_TYPE_CODE);
    WHEN (CORBA_TYPE_SHORT)
    DO;
        CALL ANYGET (EXAMPL_TEMP_ANY_ARGS.RESULT, ADDR (WS_SHORT));
        DISPLAY ('SHORT FROM ANY IS ' || WS_SHORT);
    END;
WHEN(CORBA_TYPE_LONG)
  DO;
    CALL
    ANYGET(EXAMPL_TEMP_ANY_ARGS.RESULT, ADDR(WS_LONG));
    DISPLAY('LONG FROM ANY IS ' || WS_LONG);
  END;
OTHERWISE
  DISPLAY('UNSUPPORTED TYPE IN ANY');
END;

2.16.5 ANYSET

Summary

ANYSET(PTR, /* IN : anyInfoBlock */
  PTR); /* OUT: addr(anyData) */
// Inserts data into an ANY

Description

The ANYSET utility function stores the supplied data into the ANY. ANYSET allocate
the space required for the ANY and the ORB owns the memory allocated.

Note that it is the address of the anyData that is passed to ANYSET.

Example

DCL 1 EXAMPL_TEMP_ANY_ARGS ALIGNED,
  3 RESULT PTR;
...
WS_SHORT=12;
EXAMPL_TYPE_CODE=EXAMPLE_TYPE_SHORT;

CALL TYPE-
  SET(EXAMPL_TEMP_ANY_ARGS.RESULT, EXAMPLE_TYPE_CODE, 1);
CALL ANYSET(EXAMPL_TEMP_ANY_ARGS.RESULT, ADDR(WS_SHORT));

2.16.6 MEMALOC

Summary

MEMALOC(PTR, /* OUT: pointer to memblock */
  FIXED BIN(31)); /* IN : amount of mem req'd */
// Allocates memory
Description

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

The amount of memory required is specified. If the function succeeds in allocating this number of bytes, then and the pointer is set to point to the start of this memory. If the function fails, the pointer shall contain the NULL value.

MEMALOC is used internally to allocate space for dynamic structures, as required.

Exceptions

An allocation exception shall be thrown if the memory request cannot be completed.

Example

```pli
DCL POINTR PTR;
DCL LEN FIXED BIN(31) INIT(32);
...

CALL MEMALOC(POINTR, LEN);
IF CHECK_ERRORS('MEMALOC') ^= COMPLETION_STATUS_YES THEN RETURN;
```

2.16.7 MEMDBUG

Summary

MEMDBUG(PTR, /* IN : pointer to memory */
FIXED BIN(15), /* IN : size of memory dump */
CHAR(*), /* IN : explanatory text str */
FIXED BIN(15)); /* IN : len of text string */
// Output a formatted memory dump for the specified block of memory

Description

MEMDBUG enables a developer to output a specified formatted segment of memory along with a text description. It is used for debugging purposes.

Example

```pli
CALL MEMDBUG(ADDR(MY_STRUCT), 64, 'MEMORY DUMP OF MY_STRUCT', 24);
```

Would produce a result such as the following:

```
DEBUG DUMP – MEMORY DUMP OF MY_STRUCT
00x3a598(00000): 0000E3C5 E2E340D9 C5E2E4D3 E3E20000 '"..TEST RESULTS..'
00x3a598(00010): 00E98572 009CB99A 0000FFFF 00004040 '.ZeÊ.............'
00x3a598(00020): 00000020 E2E3C1E3 C9E2E3C9 C3E20000 '"...STATISTICS..'
00x3a598(00030): 000046A2 A3998995 8700FFFF 40404000 '"...strlen9... '
```
2.16.8 MEMFREE

**Summary**

MEMFREE(PTR); /* IN: pointer to memblock */
// Frees memory

**Description**

MEMFREE is used to release dynamically allocated memory, via a pointer that was originally obtained using MEMALOC. It is also used to free dynamically allocated strings.

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**

DCL POINTR PTR;
DCL LEN FIXED BIN(31) INIT(32);

CALL MEMALOC(LEN, POINTR);
IF CHECK_ERRORS('MEMALOC') ^= COMPLETION_STATUS_YES THEN RETURN;

...

CALL MEMFREE(POINTR);

2.16.9 OBJ2STR

**Summary**

OBJ2STR(PTR, /* IN : object reference         */
        PTR); /* OUT: IOR string (null-term’d) */
// Returns an interoperable object reference (IOR)

**Description**

The OBJ2STR utility function creates an interoperable object reference (IOR) from a valid object reference.

**Example**

EXAMPLE: PROC OPTIONS(MAIN);

DCL IOR_OBJ_REF PTR;
DCL OBJECT_NAME CHAR(25) INIT('IOR:...');
DCL IOR_NAME PTR;
ALLOC POD_STATUS_INFORMATION SET(POD_STATUS_PTR);
CALL PODSTAT(POD_STATUS_INFORMATION);

CALL STRSET(IOR_NAME,OBJECT_NAME,LENGTH(OBJECT_NAME));
CALL STR2OBJ(IOR_NAME,IOR_OBJ_REF);

OBJECT_NAME=''; /* RESET OBJECT STRING */
CALL STRFREE(IOR_NAME); /* FREE THE IOR STRING */

CALL OBJ2STR(IOR_OBJ_REF,IOR_STRING);

CALL STRGET(IOR_STRING,OBJECT_NAME,LENGTH(OBJECT_NAME));
DISPLAY('THE STRINGIFIED NAME IS ' || OBJECT_NAME);
END EXAMPLE;

2.16.10 OBJGTID

Summary
OBJGTID(PTR,/* IN : object reference */
CHAR(*), /* OUT: object ID string */
FIXED BIN(31));/* IN : string length */
// Retrieves the object ID from an IOR

Description
This function retrieves the object’s ID from a given object.

Exceptions
A truncation exception shall be thrown if the object ID parameter is not large enough to hold the full length of the object ID string.

Example
DCL GRID_OBJ PTR;
DCL OBJ_ID CHAR(256) INIT('');

. . .

CALL OBJGTID(GRID_OBJ,OBJ_ID,3);
IF CHECK_ERRORS('OBJGTID') ^= COMPLETION_STATUS_YES THEN RETURN;

PUT SKIP LIST('THE OBJECT ID FOR GRID IS ',OBJ_ID);
2.16.11 OBJNEW

**Summary**

OBJNEW(CHAR(*), /* IN : server name */
       CHAR(*), /* IN : interface name */
       CHAR(*), /* IN : object ID */
       PTR); /* OUT: object reference */

// Creates a unique object reference

**Description**

This function creates a unique object reference from the supplied data. The server name is that used with the **PODINIT** function.

**Note** – All three **IN** parameters must be space-terminated.

**Example**

DCL OBJECT_REF PTR;
DCL OBJECT_ID CHAR(25);
DCL SERVER_NAME CHAR(07) INIT('SERVER ');
DCL INTERFACE_NAME CHAR(08) INIT('EXAMPLE ');

... 

OBJECT_ID='SOME_UNIQUE_VALUE';

CALL
OBJNEW(SERVER_NAME, INTERFACE_NAME, OBJECT_ID, OBJECT_REF);
CALL CHECK_ERRORS('OBJNEW');

2.16.12 OBJRIR

**Summary**

OBJRIR(PTR, /* OUT : object_ref */
       CHAR(*)); /* IN : desired_service */

// Returns an object reference to an object through which a service such as the Naming Service can be used

**Description**

The **OBJRIR** utility function shall return an object reference through which a service (for example, the Interface Repository or a CORBA service like the Naming Service) can be used.
Exceptions
A bad parameter exception shall be thrown if the service requested is not one of the following:

**IFR** _SERVICE_ for the Interface Repository
**NAMING** _SERVICE_ for the Naming Service
**TRADING** _SERVICE_ for the Trading Service

Example

```plaintext
DCL NAMING_SERVICE_OBJPTR;

/* NAMING SERVICE DECLARED IN THE CORBA COPYBOOK */
CALL OBJRIR(NAMING_SERVICE_OBJ,NAMING_SERVICE);
```

### 2.16.13 ORBARGS

**Summary**

```plaintext
ORBARGS(CHAR(*), /* IN : arg string */
FIXED BIN(31),/* IN : arg string len */
CHAR(*),/* IN : ORB name */
FIXED BIN(31))//* IN : ORB name len */

// Initializes a client or server’s connection to an ORB
```

**Description**

This function initializes a global ORB for the PL/I client that is made available then for subsequent POD calls and shall work in a similar fashion to `ORB_init`. For this reason, it must be called after the call to `PODSTAT` (assuming status information is to be made available). The argument string is used to allow environment-specific data to be passed to `ORBARGS`. If the argument list contains the ORB name as an argument, the passed in ORB name parameter shall be ignored. A program may only be initialised in one ORB, if `ORBARGS` is called more than once, the ORB made available from `ORBARGS` shall be used. Both string parameters may contain empty strings.

Note that `ORBARGS` is available to both the client and server.

**Exceptions**

If an invalid argument string or ORB name is passed to `ORBARGS`, a system exception shall be thrown.

**Example**

```plaintext
EXAMPLE: PROC(IN_ORB_ARG_STRING) OPTIONS(MAIN NOEXECOPS);

DCL IN_ORB_ARG_STRING CHAR(80) VAR;
```
DCL ORB_ARG_STRING CHAR(80) INIT(IN_ORB_ARG_STRING);
DCL ORB_NAME CHAR(20) INIT('MY_ORB');
DCL LENGTH BUILTIN;

%INCLUDE CORBA;/* CONTAINS GENERAL SETUP INFO */
ALLOC POD_STATUS_INFORMATION SET(POD_STATUS_PTR);

CALL PODSTAT(POD_STATUS_PTR);
CALL ORBARGS(ORB_ARG_STRING, LENGTH(ORB_ARG_STRING),
ORB_NAME, LENGTH(ORB_NAME));

2.16.14 PODERR

Summary

PODERR(CHAR(*), /* IN: exception string */
PTR); /* IN: addr(exception_buf) */
// Signals a user exception to the Orb

Description

PODERR informs the Orb that a user exception has occurred and enables client
programs to test for the exception. The server program must set the
exception_data if a user exception occurs and PODERR shall set the discriminator
of the union statement and the exception_id shown below. PODERR is called by a
server program. Note that this call doesn’t terminate the client or server programs, it is
up to the user to do this if required. Note also that it is the address of the
exception_buffer (i.e., the user exception buffer) that is passed to PODSTAT.
The POD shall be in charge of freeing the exception_buffer’s storage.

Note – The exception string parameter must be space terminated.

Example

Given the following IDL

interface exc {
    exception bad {
        long value;
        string<32> reason;
    };

    exception critical {
        short value_x;
        string<31> likely_cause;
    };
}
string<63> action_required;
};
}

and the following PL/I code

DCL HOST_NAME CHAR(40) INIT('');

/* THE FOLLOWING TWO DECLARATIONS ARE GENERATED BY */
/* THE GENPLI UTILITY */
DCL EXC_EXAMPL_BAD CHAR(16) INIT('EXC_EXAMPL_BAD ');
DCL 1 USER_EXCEPTIONS ALIGNED,
    3 EXCEPTION_ID PTR INIT(SYSNULL()),
    3 D FIXED BIN(31)INIT(0),
    3 U UNION,
    5 EXCEPTION_CRITICAL,
    7 VALUE_X FIXED BIN(15) INIT(0),
    7 LIKELY_CAUSE CHAR(31) INIT(''),
    7 ACTION_REQUIRED CHAR(63) INIT('');

/* OTHER DECLARATIONS */

IF HOST_NAME = '' THEN DO;
    EXCEPTION_BAD.REASON = 'NO NAME';
    EXCEPTION_BAD.VALUE1 = 99999;
    CALL PODERR(EXC_EXAMPL_BAD, ADDR(EXAMPL_USER_EXCEPTIONS));
END;

2.16.15 PODEXEC

Summary

PODEXEC(PTR, /* IN : object reference */
    CHAR(*), /* IN : operation name */
    PTR, /* INOUT: addr(op_buffer) */
    PTR); /* OUT : addr(user_exc_blk) */
// Invokes an operation on the object

Description

The PODEXEC utility function allows a PL/I client to invoke operations on the server represented by the supplied object reference. It is only available in batch mode.

The object reference passed to PODEXEC can be created using OBJSET. The operation name requested must be terminated by at least one space.
**Example**

```plaintext
DCL GRID_OBJ PTR;    /* to hold GRID’s object ref */
DCL 1 GRID_HEIGHT_ARGS, /* info about grid’s height arg */
    3 RESULT FIXED BIN(31);
DCL NO_EXCEPTION PTR INIT(SYSNULL());
DCL OPERATION_NAME CHAR(256);
...
/* STATEMENTS FOR RETRIEVING GRID’S IOR OMITTED */
...
/* TRY TO READ THE HEIGHT OF THE GRID */
OPERATION_NAME = '_get_height ';
CALL PODEXEC(GRID_OBJ,OPERATION_NAME,
    ADDR(GRID_HEIGHT_ARGS),NO_EXCEPTION);
IF CHECK_ERRORS('PODEXEC') ^= COMPLETION_STATUS_YES THEN RETURN;
PUT SKIP LIST('HEIGHT IS ',GRID_HEIGHT_ARGS.RESULT);
```

### 2.16.16 PODGET

**Summary**

PODGET(PTR); /* INOUT: addr(opArgBuffer)) */
// Gets IN and INOUT values

**Description**

Each operation implementation must begin with a call to PODGET and end with a call to PODPUT. If the operation takes no parameters and has no return value, PODGET and PODPUT must still be called passing in a dummy data area. The genpli utility generates a dummy CHAR(1) data item for this purpose.

PODGET copies the incoming operation’s argument values into the complete PL/I operation parameter buffer that is supplied. This buffer is generated automatically by the genpli utility. Only IN and INOUT values in this structure are populated by this call.

Note that is the address of the argument that is passed to PODGET.

**Example**

Consider the following IDL

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

The complete PL/I operation parameter buffer looks like:

```plaintext
DCL 1 FOO_BAR_ARGS,
    3 N FIXED BIN(15),
```
3 M FIXED BIN(15),
3 RESULT FIXED BIN(31);

The PL/I code to access this parameter list could look like:

```pli
CALL PODGET(ADDR(FOO_BAR_ARGS));
IF CHECK_ERRORS('PODGET') ^= COMPLETION_STATUS_YES THEN RETURN;

DISPLAY('N = ' || N);
FOO_BAR_ARGS.M=FOO_BAR_ARGS.N;
FOO_BAR_ARGS.RESULT=216;

CALL PODPUT(ADDR(FOO_BAR_ARGS));
IF CHECK_ERRORS('PODPUT') ^= COMPLETION_STATUS_YES THEN RETURN;
```

### 2.16.17 PODINFO

**Summary**

PODINFO(PTR); /* OUT: status info pointer */
// Retrieves address of the status_information_buffer

**Description**

PODINFO obtains the address of the status_information_buffer. If the buffer hasn’t been allocated, then it is assigned to null. Assuming that the buffer has been allocated elsewhere and followed subsequently by a call to PODSTAT, the call to PODINFO acts as if a call to PODSTAT has been made (since it recalls the address of the status_information_buffer through the status_information_pointer if used as shown below). This allows the one buffer to be used across multiple modules that shall be later linked together.

**Example**

In one PL/I module:

```pli
ALLOC POD_STATUS_INFORMATION SET(POD_STATUS_PTR);

CALL PODSTAT(POD_STATUS_PTR);
...;
```

In another PL/I module (later linked together with the module containing the above code). POD_STATUS_PTR shall now point to the same area of storage as the POD_STATUS_PTR above.

```pli
CALL PODINFO(POD_STATUS_PTR);
...;
```
2.16.18 PODINIT

Summary

PODINIT(CHAR(*), /* IN: server name */
       FIXED BIN(31)); /* IN: server name's len */
// Equivalent to impl_is_ready

Description

PODINIT is only available in batch mode. This indicates that a server is ready to start receiving requests. It is equivalent to the BOA::impl_is_ready call. The server name is passed into this call, along with its length.

Note that the server name is case-sensitive.

Example

DCL SERVER_NAME CHAR(4) INIT('grid');
DCL SNL FIXED BIN(31) INIT(4);
...

CALL PODINIT(SERVER_NAME,SNL);

2.16.19 PODPUT

Summary

PODPUT(PTR); /* INOUT: addr(opParamBuffer)) */
// Returns INOUT, OUT & result values

Description

Each operation implementation must begin with a call to PODGET and end with a call to PODPUT.

PODPUT copies the operation’s outgoing argument values from the complete PL/I operation parameter buffer passed to it. This buffer is generated automatically by the genpli utility. Only INOUT, OUT, and the special RESULT OUT item are populated by this call.

The programmer must ensure that all INOUT, OUT, and RESULT values are correctly allocated. If an exception has been raised prior to calling PODPUT, no INOUT, OUT, or RESULT parameters shall be marshalled and nothing shall be sent back in these cases, except for string truncation errors.

Note that it is the address of the argument that is passed to PODPUT.
Example

Consider the following IDL

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

The complete PL/I operation parameter buffer looks like:

```
DCL 1 FOO_BAR_ARGS ALIGNED,
    3 N FIXED BIN(15),
    3 M FIXED BIN(15),
    3 RESULT FIXED BIN(31);
```

The PL/I code to access this parameter list could look like:

```
CALL PODGET(ADDR(FOO_BAR_ARGS));
IF CHECK_ERRORS('PODGET') ^= COMPLETION_STATUS_YES THEN RETURN;

DISPLAY('N = ' || N);
FOO_BAR_ARGS.M=FOO_BAR_ARGS.N;
FOO_BAR_ARGS.RESULT=216;

CALL PODPUT(ADDR(FOO_BAR_ARGS));
IF CHECK_ERRORS('PODPUT') ^= COMPLETION_STATUS_YES THEN RETURN;
```

This returns the value of \texttt{N} back to the client in the \texttt{m} argument, and also sends the result back as the literal value \texttt{216}.

2.16.20 PODREG

Summary

```
PODREG(PTR); /* IN: interface description */
// Describes an interface to the PL/I adapter
```

Description

Each IDL interface is represented in PL/I by an include file, produced by the genpli utility. For each interface, a structure is defined and populated with information that is sufficient for subsequent calls to PODGET and PUT to be generically implemented by the POD.

Before any incoming requests can be accepted by a batch server PODREG must be called to enable the server to accept requests for that interface. PODREG is optional for CICS and IMS server implementations. PODREG must be called by the client program also in order to be able to send data to the server.
You must pass the address of the structure that is generated by the genpli utility for the interface in question. You cannot pass the structure directly due to the size of the structure that would need to be declared for PODREG as an external entry function. The format for this name shall be `<interface_name>_INTERFACE`.

**Example**

```plaintext
%INCLUDE GRIDX;
%INCLUDE CORBA; /* GENERAL SETUP INFORMATION */

...  

CALL PODREG(ADDR(GRID_INTERFACE));
IF CHECK_ERRORS('PODREG') ^= COMPLETION_STATUS_YES THEN RETURN;
```

### 2.16.21 PODREGI

**Summary**

PODREGI(PTR, /* IN : interface description*/
PTR);/* OUT: object reference */

// Describes an interface to the PL/I adapter, returning an IOR

**Description**

This function is very similar to PODREG but also returns an IOR. This can be used in conjunction with PODSRVR and OBJ2STR to retrieve the IOR string, which can then be written out to a file and be read by the client to communicate using IIOP. Care must be taken to pass in a large enough string to retrieve the IOR (a length of 1024 is sufficient). This function shall only be available in *batch* mode.

**Example**

```plaintext
DCL IORFILE FILE STREAM;
DCL IOR_NAME PTR;
DCL IORREC CHAR(1024) INIT('');
DCL IORREC_LEN FIXED BIN(31) INIT(1024);
DCL SERVER_NAME CHAR(4)  INIT('grid');
DCL SERVER_NAME_LEN FIXED BIN(31) INIT(4);
DCL SERVER_OBJ PTR;

%INCLUDE GRIDX;

CALL PODSRVR(SERVER_NAME,SERVER_NAME_LEN);
IF CHECK_ERRORS('PODSRVR') ^= COMPLETION_STATUS_YES THEN RETURN;

CALL PODREGI(ADDR(GRID_INTERFACE),SERVER_OBJ);
IF CHECK_ERRORS('PODREGI') ^= COMPLETION_STATUS_YES THEN RETURN;

OPEN FILE(IORFILE) OUTPUT;
CALL OBJ2STR(SERVER_OBJ, IOR_NAME);
CALL STRGET(IOR_NAME, IORREC, IORREC_LEN);
```
2.16.22 **PODREQ**

**Summary**

PODREQ(1, 3 PTR,/* IN: request info buffer */
  3 PTR,
  3 PTR,
  3 PTR);

// Provides current request info

**Description**

PODREQ provides information about the current invocation request, accessible via the following structure, which is defined in the CORBA include file.

```pli
DCL 1 REQUEST_INFO ALIGNED,
    3 INTERFACE_NAME PTR,
    3 OPERATION_NAME PTR,
    3 TARGET PTR;
```

The first three data items are unbounded CORBA character strings. These can be copied into CHAR(n) buffers via the STRGET function. The TARGET is a PL/I object reference for this operation invocation.

The REQUEST_INFO structure contains the following information after a call to PODREQ.

- **INTERFACE_NAME**: The name of the interface
- **OPERATION_NAME**: The name of the operation just called
- **TARGET**: The object reference (target)

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

**Example**

```pli
%INCLUDE CORBA;
...

CALL PODREQ(REQUEST_INFO);
IF CHECK_ERRORS('PODREQ') ^= COMPLETION_STATUS_YES THEN RETURN;

CALL STRGET(OPERATION_NAME,
    OPERATION,
    OPERATION_LENGTH);
```
IF CHECK_ERRORS('STRGET') ^= COMPLETION_STATUS_YES THEN RETURN;
CALL GRIDI(OPERATION);

2.16.23 **PODSRVR**

**Summary**

PODSRVR(CHAR(*), /* IN : server name */
         FIXED BIN(31)); /* IN : server name length */
// Creates the server POA

**Description**

The **PODSRVR** utility creates the server POA based on the server name supplied, via:

```pli
PortableServer::POA_ver mypoa = root_poa >create_POA("server_name",poa_manager, policies);
```

**Example**

```pli
DCL SERVER_NAME CHAR(4) INIT('grid');
DCL SERVER_NAME_LEN FIXED BIN(31) INIT(4);
...
CALL PODSRVR(SERVER_NAME,SERVER_NAME_LEN);
```

2.16.24 **PODSTAT**

**Summary**

PODSTAT(PTR); /* IN: status desc pointer */
// Registers status information block

**Description**

**PODSTAT** is used to register a supplied status information block to POD. The status of any POD call is available. This call should be made before any other POD call. Note that it is the address of the status block that is passed to **PODSTAT**.

Although **PODSTAT** is an optional call, it should always be included in every program. No status information shall be available if **PODSTAT** is not called. Also, if an exception occurs and **PODSTAT** has not been called, the program shall terminate unless **POD_STATUS_INFORMATION** has been assigned storage (this ensures that **COMPLETION_STATUS** always equals 0, i.e., ok) or if no calls to **CHECK_ERRORS** are made. Use **PODINFO** to get back the stored pointer to the **POD_STATUS_INFORMATION** data structure and to access this information.
Exceptions

PODSTAT sets EXCEPTION_NUMBER (in the exception buffer POD_STATUS_INFORMATION) to non-zero if there is an exception thrown.

POD_STATUS_INFORMATION is defined in the CORBA include file:

/*
   EXCEPTION_TEXT is a pointer to the text of the exception. STRGET must be used to extract this text.
*/
DCL POD_STATUS_PTR PTR;
DCL 1 POD_STATUS_INFORMATIONBASED(POD_STATUS_PTR),
   3 EXCEPTION_NUMBER FIXED BIN(31),
   3 COMPLETION_STATUS FIXED BIN(15),
   3 FILLER CHAR(2),
   3 EXCEPTION_TEXT PTR;
DCL COMPLETION_STATUS_YES FIXED BIN(1) INIT(0);
DCL COMPLETION_STATUS_NO FIXED BIN(1) INIT(1);
DCL COMPLETION_STATUS_MAYBE FIXED BIN(2) INIT(2);

Example

%INCLUDE CORBA;
...
ALLOC POD_STATUS_INFORMATION SET(POD_STATUS_PTR);
CALL PODSTAT(POD_STATUS_PTR);
CALL PODREG(GRID_INTERFACE);
IF CHECK_ERRORS('PODREG') ^= COMPLETION_STATUS_YES THEN RETURN;

2.16.25 SEQALOC

Summary

SEQALOC(PTR, /* OUT: sequence ctrl data */
   FIXED BIN(31), /* IN : initial seq length */
   CHAR(*), /* IN : sequence typecode */
   FIXED BIN(31)); /* IN : seq typecode len */
// Allocates storage for an unbounded sequence

Description

SEQALOC is used to allocate initial storage for an unbounded sequence. It must be called before SEQSET can be called for the first time. The length supplied to the function is the initial sequence size requested. Note that the typecode supplied to SEQALOC must be the sequence typecode.
Used On
Unbounded Sequences: ✓ Bounded Sequences: ✗

Exceptions
If the function fails in allocating sufficient space for the sequence, then an allocation exception shall be raised.

Example

```pli
dcl 1 myseq_ARGS ALIGNED,
    3 RESULT,
    5 RESULT_seq PTR,
    5 RESULT_BUF fixed bin(31);
DCL USEQLONG_TC CHAR(06) INIT('S{l},0');
...
ALLOC POD_STATUS_INFORMATION SET(POD_STATUS_PTR);
CALL PODSTAT(POD_STATUS_PTR);
CALL SEQALOC(MYSEQ_ARGS.RESULT.RESULT_SEQ,
    25,
    useqlong_tc,
    length(useqlong_tc));
IF CHECK_ERRORS('SEQALOC') ^= COMPLETION_STATUS_YES THEN RETURN;
```

2.16.26 SEQDUPL

Summary

SEQDUPL(PTR, /* IN : sequence data ptr */
    PTR); /* OUT: dupl seq data ptr */
// Duplicates an unbounded sequence control block

Description

The SEQDUPL function creates a copy of an unbounded sequence. The new sequence has the same attributes of the original sequence. The sequence data is copying into a newly allocated buffer. It is the programmer’s responsibility to release this memory.

Used On
Unbounded Sequences: ✓ Bounded Sequences: ✗

Example

```pli
DCL 1 EXAMPL_SEQ_ARGS ALIGNED,
    3 RESULT,
    5 RESULT_SEQ PTR,
```
DCL 1 EXAMPL_SEQ_2_ARGS ALIGNED,
  3 RESULT,
  5 RESULT_SEQ PTR,
  5 RESULT_BUF FLOAT DEC(6);

DCL ELEMENT_NUM FIXED BIN(31);
DCL MAX_SEQ_ELE FIXED BIN(31);

... 

CALL SEQMAX(EXAMPL_SEQ_ARGS.RESULT.RESULT_SEQ,MAX_SEQ_ELE);

DO ELEMENT_NUMBER = 1 TO MAX_SEQ_ELE;
  CALL PROCESS_INIT_SEQUENCE_ENTRY;
  CALL SEQSET(EXAMPL_SEQ_2_ARGS.RESULT_1.RESULT_SEQ,
               ELEMENT_NUM,VECTOR);
END;

CALL SEQDUPL(EXAMPL_SEQ_ARGS.RESULT_1.RESULT_SEQ,
              EXAMPL_SEQ_2_ARGS.RESULT_1.RESULT_SEQ);

...

### 2.16.27 SEQFREE

#### Summary

SEQFREE(PTR); /* IN: sequence data ptr */
// Frees an unbounded sequence

#### Description

SEQFREE is used to release storage assigned to a sequence via SEQALLOC and SEQINIT.

Care should be taken not to attempt to de-reference this pointer after freeing it, as this may result in a run-time error. It is important to SEQFREE from the innermost nested sequence to the outermost or memory leaks shall occur as a result.

#### Used On

Unbounded Sequences: ✓  Bounded Sequences: ✗

#### Example

dcl 1 myseq_ARGS ALIGNED,
  3 RESULT,
```pli
5 RESULT_seq
5 RESULT_Buf

DCL USEQLONG_TC
    CHAR(06)       INIT('S{1},0');

CALL SEQALOC(MYSEQ_ARGS.RESULT.RESULT_SEQ,
    25,
    useqlong_tc,
    length(useqlong_tc));
IF CHECK_ERRORS('SEQALOC') ^= COMPLETION_STATUS:YES THEN RETURN;

2.16.28 SEQGET

Summary
SEQGET(PTR, /* IN : sequence data ptr */
    FIXED BIN(31), /* IN : element number */
    PTR); /* OUT: addr(seq_buffer) */
// Retrieves the element_number element of an unbounded sequence

Description
The SEQGET utility function provides access to a specific element of an unbounded sequence. The data is copied from the specified element into the supplied data area (i.e., the ‘seq_buffer’).

Used On
Unbounded Sequences: ✓ Bounded Sequences: ✗

Exceptions
A bounds exception shall be thrown if an attempt is made to get any element greater than the current length of the sequence.

Example
DCL 1 EXAMPL_SEQ_ARGS ALIGNED,
    3 RESULT,
    5 RESULT_SEQ
    5 RESULT_BUF
    DCL ELEMENT_NUM
    FIXED BIN(31);
    DCL NUM_SEQ_ELE
    FIXED BIN(31);

...
CALL SEQLEN(EXAMPL_SEQ_ARGS.RESULT.RESULT_SEQ,NUM_SEQ_ELE);

DO ELEMENT_NUM = 1 TO NUM_SEQ_ELE;
    CALL SEQGET(RESULT_SEQ,ELEMENT_NUM,ADDR(RESULT_BUF));
    IF CHECK_ERRORS('SEQGET') ^= COMPLETION_STATUS_YES THEN RETURN;

    CALL PROCESS_SEQUENCE_ENTRY;
END;

### 2.16.29 SEQINIT

#### Summary

SEQINIT(PTR, /* OUT: sequence data ptr */
        CHAR(*), /* IN : sequence typecode */
        FIXED BIN(31)); /* IN : seq typecode len */

// Initializes a bounded sequence

#### Description

SEQINIT is used initialize a bounded sequence. It sets the maximum and current length to the size of the bounded sequence and the sequence typecode to that supplied to SEQINIT. The buffer is set to NULL. If the user wishes to fill only part of the sequence, SEQLSET can be used to indicate how many items of the sequence have been filled. It is important to note that it is the sequence typecode which must be supplied to SEQINIT.

#### Used On

Unbounded Sequences: ✗ Bounded Sequences: ✓

#### Exceptions

If an invalid typecode is passed to SEQINIT, a system exception shall be raised. If an initialized sequence data area is passed in, the existing data area is passed back.

#### Example

dcl 1 myseq_ARGS ALIGNED,
    3 RESULT,
        5 RESULT_seq           PTR,
        5 RESULT_DAT(10) fixed bin(31);

DCL SEQLONG10_TC CHAR(07) INIT('S1,10');

... 

ALLOC POD_STATUS_INFORMATION SET(POD_STATUS_PTR);
CALL PODSTAT(POD_STATUS_PTR);

CALL SEQINIT(MYSEQ_ARGS.RESULT.result_seq,
               seqlong10_tc,
2.16.30 SEQLEN

Summary

SEQLEN(PTR, /* IN : sequence data ptr */
       FIXED BIN(31)); /* OUT: length of sequence */
// Retrieves the current length of the sequence

Description

The SEQLEN utility function retrieves the current length of a given bounded or unbounded sequence. The out parameter len should be stored as a FIXED BIN(31) variable.

Example

A BAD_SEQ exception is returned if a null pointer is supplied to SEQLEN.

Used On

Unbounded Sequences: ✓ Bounded Sequences: ✓

Example

dcl 1 myseq_args ALIGNED,
    3 result,
    5 result_seq PTR,
    5 result_BUF fixed bin(31);
dcl element_num fixed bin(31);
DCL NUM_SEQ_ELE FIXED Bin(31) init(0);
...

CALL SEQLEN(MYSEQ_ARGS.RESULT.RESULT_SEQ,NUM_SEQ_ELE);

do element_num = 1 to NUM_SEQ_ELE;
call display_seq_info(MYSEQ_ARGS.RESULT.RESULT_SEQ,element_num);
end;
### 2.16.31 SEQLSET

**Summary**

```plaintext
SEQLSET(PTR, /* IN: sequence data ptr */
       FIXED BIN(31)); /* IN: new length of seq */
// Changes the number of elements in the sequence
```

**Description**

The `SEQLSET` utility function is used to resize the sequence. The `new_length` of the sequence can be any amount from 1 to the current length of the sequence plus one (but not larger than the maximum length for the sequence). Note that if a sequence is made smaller, then the contents of the elements greater than the new length of the sequence are undefined. This function can be used for both bounded and unbounded sequences. The function is used with unbounded sequences generally for restricting access to a subset of the total sequence.

**Used On**

- Unbounded Sequences: ✓
- Bounded Sequences: ✓

**Exceptions**

A bounds exception shall be thrown if an attempt is made to set any element greater than the current length of the sequence plus one (or greater than the maximum length defined for the sequence).

If a NULL sequence is passed to `SEQLSET`, a `BAD_SEQ` exception shall be set.

**Example**

```plaintext
DCL 1 EXAMPL_MYSEQ_ARGS ALIGNED,
       3 RESULT, 
       5 RESULT_SEQ PTR, 
       5 RESULT_BUF FLOAT DEC(6); 

DCL ELEMENT_NUMFIXED BIN(31); 

... 

SEQLSET(EXAMPL_MYSEQ_ARGS.RESULT.RESULT_SEQ,5); 
IF CHECK_ERRORS('SEQLSET') ^= COMPLETION_STATUS_YES THEN RETURN; 
```

### 2.16.32 SEQMAX

**Summary**

```plaintext
SEQMAX(PTR, /* IN: sequence data ptr */
       FIXED BIN(31)); /* OUT: max len of sequence */
// Returns the maximum set length of the sequence
```
Description

The SEQMAX utility function retrieves the current maximum length of a given bounded or unbounded sequence. In the case of a bounded sequence, this would be set to the bounded size. In unbounded sequences, this is at least the size of the initial number of elements declared for the unbounded sequence (e.g., through SEQALLOC). The out parameter max should be stored as a FIXED BIN(31) variable.

Used On

Unbounded Sequences: ✓  Bounded Sequences: ✓

Exceptions

A BAD_SEQ exception is returned if a null pointer is supplied to SEQMAX.

Example

dcl 1 myseq_ARGS ALIGNED,
    3 result,
    5 result_seq PTR,
    5 result_BUF fixed bin(31);

DCL USEQLONG_TC CHAR(06) INIT('S{l},0');
DCL MYSEQ_maxLENGTH FIXED BIN(31) init(0);

... CALL SEQALLOC(MYSEQ_ARGS.RESULT.RESULT_SEQ, 25, useqlong_tc, length(useqlong_tc));
IF CHECK_ERRORS('SEQALLOC') ^= COMPLETION_STATUS_YES THEN RETURN;
SEQMAX(MYSEQ_ARGS.RESULT.RESULT_SEQ, MYSEQ_LENGTH);
CALL CHECK_ERRORS('SEQMAX');
DISPLAY('INITIAL MAXIMUM SEQUENCE LENGTH = ' || MYSEQ_LENGTH);

2.16.33 SEQSET

Summary

SEQSET(PTR,           /* IN : sequence data ptr */
    FIXED BIN(31),     /* IN : element number */
    PTR);              /* IN : addr(seq_buffer) */
// Stores the data into the element_number element of
// an unbounded sequence.
**Description**

The `SEQSET` utility function stores the supplied data into the requested element of an unbounded sequence. If the requested element number is greater than the current length, the size of the sequence shall be increased to accommodate the requested element. If the current maximum element plus one is set then the sequence shall get reallocated to hold the enlarged sequence.

**Used On**

Unbounded Sequences: ✓ Bounded Sequences: ✗

**Exceptions**

An exception shall be thrown if an attempt is made to set any element greater than the current length of the sequence plus one.

**Example**

```plaintext
DCL 1 MYSEQ_ARGS ALIGNED,
    3 RESULT,
    5 RESULT_SEQ PTR,
    5 RESULT_BUF FLOAT DEC(6);

DCL ELEMENT_NUM FIXED BIN(31);
DCL MAX_SEQ_ELE FIXED BIN(31);

... 

CALL SEQMAX(MYSEQ_ARGS.RESULT.RESULT_SEQ,MAX_SEQ_ELE);

DO ELEMENT_NUM = 1 TO MAX_SEQ_ELE;
    CALL PROCESS_INIT_SEQUENCE_ENTRY;
    CALL SEQSET(MYSEQ_ARGS.RESULT.RESULT_SEQ,
                ELEMENT_NUM,
                ADDR(MYSEQ_ARGS.RESULT.RESULT_BUF));
    IF CHECK_ERRORS('SEQSET') ^= COMPLETION_STATUS_YES THEN RETURN;
END;
```

### 2.16.34 STR2OBJ

**Summary**

`STR2OBJ(PTR, /* IN : IOR string (null-terminated) */
        PTR); /* OUT: object reference */`

// Creates an object reference from an IOR

**Description**

The `OBJSET` utility function creates an object reference from an IOR.
Example

DCL GRID_OBJ PTR;
DCL OBJECT_NAME CHAR(64) INIT('IOR:...');
DCL IOR_NAME PTR;

CALL STRSET(IOR_NAME, OBJECT_NAME, LENGTH(OBJECT_NAME));
CALL STR2OBJ(IOR_NAME, GRID_OBJ);
IF CHECK_ERRORS('OBJSET') ^= COMPLETION_STATUS_YES THEN RETURN;

2.16.35 STRCON

Summary

STRCON(PTR, /* INOUT: string pointer */
       PTR); /* IN : 'addon' string ptr */
// Concatenates two unbounded strings

Description

The STRCON utility function concatenates the two supplied unbounded strings, returning the concatenated unbounded string in str_pointer1. Note that the original storage allocated to str_pointer1 shall be deleted as it shall be assigned the concatenated string instead.

Example

DCL FIRST_PART PTR;
DCL SECOND_PART PTR;
.
.
.

CALL STRCON(FIRST_PART, SECOND_PART);

2.16.36 STRDUPL

Summary

STRDUPL(PTR, /* IN : string pointer */
        PTR); /* OUT: duplicate string pointer */
// Duplicates a given unbounded string

Description

The STRDUPL utility function takes in an unbounded string str_pointer and duplicates the string, returning the duplicate via dupl_pointer. This is a complete copy (i.e., the storage used by str_pointer is also duplicated).
Example

DCL ORIG_STR_PTR PTR;
DCL DUPL_STR_PTR PTR;

CALL STRDUPL(ORIG_STR_PTR, DUPL_STR_PTR);

2.16.37 STRFREE

Summary

STRFREE(PTR); /* IN: unbounded string pointer */
// Frees the storage of an unbounded string

Description

STRFREE is used to release the storage of an unbounded string.

Example

DCL MY_STRING CHAR(50) INIT('HELLO');
DCL MY_UNB_STRING PTR;

CALL STRSET(MY_UNB_STRING, MY_STRING, LENGTH(MY_STRING));
IF CHECK_ERRORS('STRSET') ^= COMPLETION_STATUS_YES THEN RETURN;
...

CALL STRFREE(MY_UNB_STRING); /* FINISHED WITH UNBOUNDED STRING */

2.16.38 STRGET

Summary

STRGET(PTR, /* IN : string pointer */
CHAR(*), /* OUT : PL/I string */
FIXED BIN(31)); /* IN : PL/I string length */
// Copies the contents of a dynamic string to a CHAR(n) data item

Description

This utility function copies the characters in the unbounded string pointer
src_pointer to the dest PL/I CHAR(dest_len) string item. If the
src_pointer does not contain enough characters to exactly fill the dest, then it
shall be space padded. If there are too many characters in the src_pointer for
dest, then only dest_len characters from the string pointer gets copied over and a
truncation exception shall be raised.

NULL characters shall never be copied from the src_pointer to the dest.
Exceptions
A truncation exception gets raised if the length of the source is greater than the given destination.

Example
/* This is the supplied PL/I unbounded string pointer */
DCL SRC_POINTER PTR;

/* This is the PL/I representation of the string */
DCL DEST CHAR(64);
DCL DEST_LEN FIXED BIN(31) INIT(LENGTH(DEST));

/* This STRGET call copies the characters in the NAME */
/* to the SUPPLIER_NAME */
CALL STRGET(SRC_POINTER,DEST,DEST_LEN);
IF CHECK_ERRORS('STRGET') ^= COMPLETION_STATUS_YES THEN RETURN;

2.16.39 STRLENG

Summary
STRLENG(PTR /* IN: string pointer */
FIXED BIN(31)); /* OUT: len of string */
// Returns the actual length of an unbounded string

Description
The STRLENG utility function return the number of characters in an unbounded string.

Example
DCL STR_PTR PTR;
DCL LEN FIXED BIN(31);

CALL STRLENG(STR_PTR,LEN);

2.16.40 STRSET, STRSETS

Summary
STRSET (PTR, /* OUT: string ptr - no pad */
CHAR(*), /* IN: PL/I string */
FIXED BIN(31)); /* IN: PL/I string length */
STRSETS (PTR, /* OUT: string ptr with pad */
CHAR(*), /* IN: PL/I string */
FIXED BIN(31)); /* IN: PL/I string length */
// Creates a dynamic string from a CHAR(n) data item
Description

The **STRSET** utility function creates an unbounded string and copies `src_length` characters from `src` to `dest_pointer`. If `src` contains trailing spaces these shall not be copied to the dest string.

The **STRSETS** version of this function is identical, except it shall copy trailing spaces.

Example

```c
/* This is the 'source' CHAR(n) data item */
DCL SRC CHAR(160);
DCL SRC_LENGTH FIXED BIN(31) INIT(LENGTH(SRC));

/* This is the 'destination' unbounded pointer string */
DCL DEST_PTR PTR;

/* This STRSET call creates a copy of the string in the */
/* SRC field and assigns the pointer to DEST_PTR. */
CALL STRSET(DEST_PTR, SRC, SRC_LENGTH);
```

### 2.16.41 TYPEGET

#### Summary

```c
TYPEGET(PTR, /* IN : addr(anyInfoBlock) */
        CHAR(*), /* OUT: typecode */
        FIXED BIN(31)); /* IN : typecode length */
```

// Extracts type name out of an ANY

#### Description

The **TYPEGET** utility function returns the type code of the **ANY**. This function can be used to get the type of the **ANY** so that the correct buffer is passed to the **ANYGET** function.

#### Exceptions

A truncation exception shall occur if the length of the **typeName** passed to **TYPEGET** is too small to contain the type code stored in the **ANY**.

#### Example

```c
DCL 1 EXAMPL_TEMP_ANY_ARGS ALIGNED,
     3 RESULT PTR;

DCL WS_SHORT FIXED BIN(15);
DCL WS_LONG FIXED BIN(31);
```
CALL TYPEGET(EXAMPL_TEMP_ANY_ARGS.RESULT, 
    EXAMPL_TYPE_CODE, 
    EXAMPL_TYPE_CODE_LENGTH); 
IF CHECK_ERRORS('TYPEGET') ^= COMPLETION_STATUS_YES THEN RETURN;

SELECT(EXAMPL_TYPE_CODE);
    WHEN(CORBA_TYPE_SHORT) 
        DO;
            CALL ANYGET(EXAMPL_TEMP_ANY_ARGS.RESULT, ADDR(WS_SHORT)); 
            IF CHECK_ERRORS('ANYGET') ^= COMPLETION_STATUS_YES THEN RETURN;

            DISPLAY('SHORT FROM ANY IS ' || WS_SHORT);
        END;
    WHEN(CORBA_TYPE_LONG) 
        DO;
            CALL ANYGET(EXAMPL_TEMP_ANY_ARGS.RESULT, ADDR(WS_LONG)); 
            IF CHECK_ERRORS('ANYGET') ^= COMPLETION_STATUS_YES THEN RETURN;

            DISPLAY('LONG FROM ANY IS ' || WS_LONG);
        END;
    OTHERWISE 
        DISPLAY('UNSUPPORTED TYPE IN ANY');
END;

2.16.42 TYPESET

Summary

TYPESET(PTR, /* IN: addr(anyInfoBlock) */ 
    CHAR(*), /* IN: typecode */ 
    FIXED BIN(31)); /* IN: typecode length */
// Sets the type name of an ANY

Description

The TYPESET utility function sets the type of the ANY to the supplied typecode. This should be done prior to calling ANYSET as ANYSET uses the current typecode information to insert the data into the ANY.

Example

DCL 1 EXAMPLE_TEMP_ANY_ARGS ALIGNED, 
    3 RESULT PTR;

    ... 

WS_SHORT=12;
EXAMPL_TYPE_CODE=CORBA_TYPE_SHORT;
2.16.43 WSTRCON

Summary

WSTRCON(PTR, /* INOUT: wide string pointer */
        PTR); /* IN : 'addon' wide string ptr */
// Concatenates two unbounded strings

Description

The WSTRCON utility function concatenates the two supplied unbounded wide strings, returning the concatenated unbounded wide string in the first parameter. Note that the original storage allocated to the first parameter shall be deleted as it shall be assigned the concatenated string instead.

Example

DCL FIRST_PART PTR;
DCL SECOND_PART PTR;

. . .

CALL WSTRCON(FIRST_PART, SECOND_PART);

2.16.44 WSTRDUP

Summary

WSTRDUP(PTR, /* IN : string pointer */
         PTR); /* OUT: duplicate string pointer */
// Duplicates a given unbounded wide string

Description

The WSTRDUP utility function takes in an unbounded string str_pointer and duplicates the string, returning the duplicate via dupl_pointer. This is a complete copy (i.e., the storage used by both wide strings shall be duplicated).

Example

DCL ORIG_WIDESTR_PTR PTR;
DCL DUPL_WIDESTR_PTR PTR;
CALL WSTRDUP(ORIG_STR_PTR, DUPL_STR_PTR);

2.16.45 WSTRFRE

Summary
WSTRFRE(PTR); /* IN: unbounded wide string pointer */
// Frees the storage of an unbounded wide string

Description
WSTRFRE is used to release the storage of an unbounded wide string.

Example
DCL MY_WSTRING WIDECHAR(50) INIT('HELLO');
DCL MY_UNB_WSTRING PTR;
CALL WSTRSET(MY_UNB_WSTRING, MY_WSTRING, LENGTH(MY_WSTRING));
IF CHECK_ERRORS('WSTRSET') ^= COMPLETION_STATUS_YES THEN RETURN;
...
CALL WSTRFRE(MY_UNB_WSTRING);

2.16.46 WSTRGET

Summary
WSTRGET(PTR, /* IN : wide string pointer */
    WIDECHAR(*), /* OUT: PL/I wide string */
    FIXED BIN(31)); /* IN : PL/I w-string length */
// Copies the contents of an unbounded wide string to a
// WIDECHAR(n) data item

Description
This utility function copies the characters in the unbounded wide string pointer to the
PL/I WIDECHAR(wstring_len) string item. If the wide string pointer’s contents
does not contain enough characters to exactly fill the PL/I wide string then it shall be
space padded. If there are too many characters in the wide string pointer’s contents for
the PL/I wide string then only the length of the PL/I wide string shall get copied from
the string pointer gets and a truncation exception shall be raised.

NULL characters shall never be copied from the src_pointer to the dest.
Exceptions
A truncation exception gets raised if the length of the source is greater than the given destination.

Example

/* This is the supplied PL/I unbounded string pointer */
DCL SRC_POINTER PTR;

/* This is the PL/I representation of the string */
DCL DEST_WIDESTR WIDECHAR(64);
DCL DEST_LEN FIXED BIN(31) INIT(LENGTH(DEST_WIDESTR));

/* This WSTRGET call copies the characters in the NAME */
/* to the SUPPLIER_NAME */
CALL WSTRGET(SRC_POINTER, DEST_WIDESTR, DEST_LEN);
IF CHECK_ERRORS('WSTRGET') ^= COMPLETION_STATUS_YES THEN RETURN;

2.16.47 WSTRLLEN

Summary

WSTRLLEN(PTR /* IN : wstring pointer */
     FIXED BIN(31)); /* OUT: len of wstring */
// Returns the length of an unbounded wide string

Description

The WSTRLLEN utility function returns the number of characters in an unbounded wide string.

Example

DCL WIDE_STR_PTR PTR;
DCL LEN FIXED BIN(31);

CALL WSTRLLEN(WIDE_STR_PTR, LEN);

2.16.48 WSTRSET, WSTRSTS

Summary

WSTRSET(PTR, /* OUT: wstring ptr - no pad */
     CHAR(*), /* IN : PL/I wide string */
     FIXED BIN(31)); /* IN : PL/I w-string length */
WSTRSTS(PTR, /* OUT: wstring ptr with pad */
     CHAR(*), /* IN : PL/I wide string */
     FIXED BIN(31)); /* IN : PL/I w-string length */
// Creates an unbounded wide string from a WIDECHAR(n) data item
Description

The `WSTRSET` utility function creates an unbounded string and copies `src_length` characters from `src` to `dest_pointer`. If `src` contains trailing spaces these shall not be copied to the `dest` string.

The `WSTRSTS` version of this function is identical, except it shall copy trailing spaces.

Example

```c
/* This is the 'source' WIDECHAR(n) data item */
DCL SRC WIDECHAR(160);
DCL SRC_LENGTH FIXED BIN(31) INIT(LENGTH(SRC));
/* This is the 'destination' unbounded pointer wide string */
DCL DEST_PTR PTR;

/* This WSTRSET call creates a copy of the wide string */
/* in the SRC field and assigns the pointer to DEST_PTR. */
CALL WSTRSET(DEST_PTR, SRC, SRC_LENGTH);
```
A.1 Summary

CHECK_ERRORS(CHAR(*)) RETURNS(FIXED BIN(31)); /* IN: procedure called */
/* RETURNS: COMP_STATUS */
// Used to test for errors after a POD call

A.2 Description

This is not a POD function as such but is a function that shall be used in POD programming examples to ensure that calls to the POD functions in a given program execute as expected. This function shall contain any user exceptions that also need to be tested for. The function is IDL-dependent in that it generates a separate WHEN clause for each user exception defined.

The following example demonstrates the layout of CHECK_ERRORS with two user exceptions specifically mentioned (exceptions 'bad' and 'worse' in interface 'exampl') and what the function does in the case of a system exception. It is good practice to include a call to CHECK_ERRORS after every POD call.

Note that this function shown below is a sample implementation of CHECK_ERRORS and demonstrates how to check for system exceptions and user exceptions.

A.3 Example

CHECK_ERRORS: PROC(FUNCTION_NAME) RETURNS(FIXED BIN(15));
dcl function_name char(*);
dcl sysprint ext file stream print output;
dcl exception_number fixed bin(31) init(0);
dcl exception_info char(*) ctl;
dcl exception_len fixed bin(31) init(0);
dcl exc_name char(78) init('');
dcl return_code fixed bin(15) init(0);

call podinfo(pod_status_ptr);

exception_number=pod_status_information.exception_number;

if exception_number = pod_user_exception then
  do;
    call strget(exception_id,78,exc_name);

    put skip list('EXAMPL_USER_EXCEPTION encountered');
    put skip list('Function Called:',function_name);
    put skip list('Return Code    :',exception_number);

    select(exc_name);
      when('exc_exampl_bad') /* user exception 'bad' */
        do;
          /* Display any error settings available */
          end;
      when('exc_exampl_worse') /* user exception 'worse' */
        do;
          /* Display any error settings available */
          end;
      otherwise;
        end;
    return_code=completion_status_no;
  end;
else if exception_number ^= 0 then
  do;
    call strleng(exception_text,exception_len);
    alloc exception_info char(exception_len);
    call strget(exception_text,exception_info,exception_len);

    put skip list('SYSTEM EXCEPTION');
    put skip list('Function Called:',function_name);
    put skip list('Exception    :',exception_info);
    put skip list('Return Code    :',exception_number);
    return_code=completion_status_no;
    free exception_info;
  end;
else
  return_code=completion_status_yes;

END CHECK_ERRORS;
Index

A
Abstract IDL valuetypes 2-22
ALIGNED 2-6
any type 2-10
ANYFREE 2-45
ANYGET 2-10, 2-46
ANYSET 2-10, 2-47
Application Servants 2-30
array definition 2-9

B
Basic IDL types 2-2
boolean 2-3
Bounded sequences 2-7
bounded string 2-4

C
CHAR data values 2-3
CHECK_ERRORS A-1
compliance iii
CORBA
    contributors iv
documentation set ii
core, compliance iv

D
DISPTCH 2-43

E
EPV 2-28
exception 2-11

F
fixed type 2-5

I
Identifiers 1-2
INOUT 2-44
interface 2-13
interoperability, compliance iv
interworking
    compliance iv

M
Mapping for Anys 2-10
Mapping for Arrays 2-9
Mapping for Basic Types 2-2
Mapping for Exceptions 2-11
Mapping for Fixed Types 2-5
Mapping for Interfaces 2-13
Mapping for Sequence Types 2-7
Mapping for Strings 2-4
Mapping for Struct Types 2-6
Mapping for Typedefs 2-13
Mapping for Union Types 2-6
Mapping for Valuetypes 2-14
MEMALLOC 2-48
MEMDBG 2-48
MEMFREE 2-29, 2-35, 2-49
Memory 2-33, 2-35

O
OBJ2STR 2-42, 2-49
Object Management Group i
    address of iii
OBJGTID 2-50
OBJNEW 2-51
OBJRIR 2-51
Octet data type 2-3
ORBARGS 2-42, 2-52

P
PODERR 2-12, 2-53
PODEXEC 2-54
PODGCT 2-8
PODINFO 2-56
PODINIT 2-43, 2-57
PODPUT 2-9, 2-35, 2-44, 2-57
PODREG 2-58
PODREGI 2-42
PODREQ 2-43, 2-60
PODSVR 2-61
PODSTAT 2-42, 2-61
Portable Object Adapter Mapping 2-24
PortableServer
    Servant Mapping 2-26
    PortableServer Procedures 2-25

S
SELECT 2-7
SEQALLOC 2-8, 2-9, 2-34, 2-62
SEQDUPL 2-63
SEQFREE 2-34, 2-64
SEQGET 2-8, 2-65
SEQINIT 2-66
SEQLEN 2-7, 2-8, 2-67
SEQLSET 2-68
SEQMAX 2-7, 2-8, 2-69
SEQSET 2-8, 2-9, 2-34, 2-70
Server implementation 2-42
Skeletons 2-27
STR2OBJ 2-70
STRCON 2-71
STRDUPL 2-71
STRFREE 2-72
STRGET 2-72
string data type 2-4
STRLENG 2-35, 2-73
STRSET 2-5, 2-35, 2-73
STRSETS 2-73

T
typedefs 2-13
TYPEGET 2-10, 2-46, 2-74
TYPESET 2-75

U
Unbounded sequences 2-8
unbounded string 2-4
UNION 2-7
union 2-6
Index

V
VALBASE 2-19
Value boxes 2-20
VALUEBASE 2-16
Valuetype inheritance 2-22
valuetype members 2-14
Valuetype operations 2-16
Valuetype Procedures 2-19
Valuetypes 2-14

W
WIDECCHAR 2-77
WSTRCON 2-76
WSTRDUP 2-76
WSTRFRE 2-77
WSTRGET 2-77
WSTRLEN 2-78
WSTRSET 2-78
WSTRSTS 2-78
This is the first formal version of the PL/I Language Mapping specification.

OMG documents used to create this version:

- FTF Final Adopted specification: ptc/01-01-01
- FTF Report: ptc/01-04-01